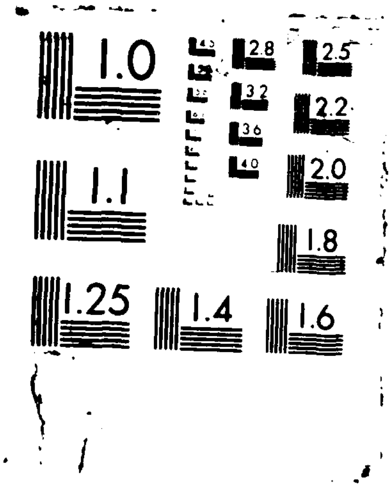
AD-A190 166 NEW GENERATION KNOWLEDGE PROCESSING VOLUME 2(U) 1/3
SYRACUSE UNIV NY J A ROBINSON ET AL. OCT 87
RADC-TR-87-165-VOL-2 F30602-84-K-0001

UNCLASSIFIED F/G 12/5 NL


1.0
2.8
2.5
3.2
2.2
3.6
4.0
2.0
1.1
1.8
1.25
1.4
1.6

RADC-TR-87-165, Vol II (of three)
Final Technical Report
October 1987

NEW GENERATION KNOWLEDGE PROCESSING

Syracuse University

J. Alan Robinson and Kevin J. Greene

APPROVED FOR PUBLIC RELEASE; DISTRIBUTION UNLIMITED.

ROME AIR DEVELOPMENT CENTER
Air Force Systems Command
Griffiss Air Force Base, NY 13441-5700

This report has been reviewed by the RADC Public Affairs Office (PA) and is releasable to the National Technical Information Service (NTIS). At NTIS it will be releasable to the general public, including foreign nations.

RADC-TR-87-165, Vol II (of three) has been reviewed and is approved for publication.

APPROVED: Northrup Fowler

NORTHRUP FOWLER III
Project Engineer

APPROVED: Raymond P. Urtz Jr

RAYMOND P. URTZ, JR.
Technical Director
Directorate of Command & Control

FOR THE COMMANDER: Richard W. Pouliot

RICHARD W. POULIOT
Directorate of Plans & Programs

If your address has changed or if you wish to be removed from the RADC mailing list, or if the addressee is no longer employed by your organization, please notify RADC (COES) Griffiss AFB NY 13441-5700. This will assist us in maintaining a current mailing list.

Do not return copies of this report unless contractual obligations or notices on a specific document require that it be returned.

UNCLASSIFIED
SECURITY CLASSIFICATION OF THIS PAGE

REPORT DOCUMENTATION PAGE

Form Approved
OMB No 0704-0188

1a REPORT SECURITY CLASSIFICATION
UNCLASSIFIED

1b RESTRICTIVE MARKINGS
N/A

2a SECURITY CLASSIFICATION AUTHORITY
N/A

2b DECLASSIFICATION DOWNGRADING SCHEDULE
N/A

3 DISTRIBUTION/AVAILABILITY OF REPORT
Approved for public release;
distribution unlimited

4 PERFORMING ORGANIZATION REPORT NUMBER(S)
N/A

5 MONITORING ORGANIZATION REPORT NUMBER(S)
RADC-TR-87-165, Vol II (of three)

6a NAME OF PERFORMING ORGANIZATION
Syracuse University

6b OFFICE SYMBOL (If applicable)

7a NAME OF MONITORING ORGANIZATION
Rome Air Development Center (COES)

6c ADDRESS (City, State and ZIP Code)
Syracuse NY 13244

7b ADDRESS (City, State, and ZIP Code)
Griffiss AFB NY 13441-5700

8a NAME OF FUNDING SPONSORING ORGANIZATION
Rome Air Development Center

8b OFFICE SYMBOL (If applicable)
COES

9 PROCUREMENT INSTRUMENT IDENTIFICATION NUMBER
F30602-84-K-0001

8c ADDRESS (City, State, and ZIP Code)
Griffiss AFB NY 13441-5700

10 SOURCE OF FUNDING NUMBERS

PROGRAM ELEMENT NO	PROJECT NO	TASK NO	WORK UNIT ACCESSION NO
62702F	5581	27	10

11 TITLE (Include Security Classification)
NEW GENERATION KNOWLEDGE PROCESSING

12 PERSONAL AUTHOR(S)
J. Alan Robinson, Kevin J. Greene

13a TYPE OF REPORT
Final

13b TIME COVERED
FROM Dec 83 TO Jan 87

14 DATE OF REPORT (Year, Month, Day)
October 1987

15 PAGE COUNT
200

16 SUPPLEMENTARY NOTATION
N/A

17 COSATI CODES

FIELD	GROUP	SUB GROUP
12	05	

18 SUBJECT TERMS (Continue on reverse if necessary and identify by block number)
Artificial Intelligence, Graph Reduction,
Logic Programming, Combinators,
Functional Programming, Programming Languages

19 ABSTRACT (Continue on reverse if necessary and identify by block number)

The main goal of this project was to design a high-level programming system (which we have named SUPER, an acronym for "Syracuse University Parallel Expression Reducer") with two parts: a language which would combine the functional (as in LISP, SASL or ML) with the relational (as in PROLOG) programming concepts into a single new paradigm and a machine which would execute programs written in the language, using reduction and a multiprocessor architecture.

The SUPER language is an extension of the basic lambda-calculus which we call lambda plus. It is formally a collection of expressions together with some rules and definitions which give them meaning and make it possible to do deductive reasoning and computation with them. The expressions of the SUPER language fall into three main syntactic categories: atoms, abstractions, and combinations.

Volume I describes the SUPER system, and discusses the conceptual background in terms of (over)

20 DISTRIBUTION AVAILABILITY OF ABSTRACT
☒ UNCLASSIFIED/UNLIMITED ☐ SAME AS RPT ☐ DTIC USERS

21 ABSTRACT SECURITY CLASSIFICATION
UNCLASSIFIED

22a NAME OF RESPONSIBLE INDIVIDUAL
Northrup Fowler III

22b TELEPHONE (Include Area Code)
(315) 330-7794

22c OFFICE SYMBOL
RADC (COES)

Previous editions are obsolete

SECURITY CLASSIFICATION OF THIS PAGE
UNCLASSIFIED

Block 19. Abstract (Cont'd)

which it can best be understood. In developing these ideas over the period of the project we devised and implemented two related single-processor reduction systems, LNF and LNF-Plus, as experimental tools to help us learn more about SUPER language design issues. These systems have turned out to be of considerable interest and utility in their own right, and they have taken on separate and independent identities.

Volume 2 contains a detailed presentation of the single-processor software programming system LNF which was developed to serve as a test bed and simulation tool for the "classical" part of the SUPER system.

Volume 3 presents the final, enhanced version of LNF, which we call LNF-Plus and which provides the user with as close an approximation as we can achieve on a single processor of the SUPER system. Volume 3 is also designed as a useful guide to someone who wishes to use the system for experimental computations.

Abstract

In the first third of this thesis, three well known reduction calculi: A. Church's λ-calculus, M. Schönfinkel's SKI-calculus, and C.P. Wadsworth's graph oriented λ-calculus (λ-G-calculus) are defined. Schönfinkel's classic transformation of λ-calculus well-formed formulas (wffs) into variable-free SKI-calculus wffs is also presented. A new notion, lazy-normal form, a generalization of the SKI-calculus' concept of normal form, is then defined and compared with Wadsworth's concept of head-normal form. Head-normal form is a generalized notion of normal form in the λ-calculus. It is demonstrated that an SKI-calculus wff in lazy-normal form is an outline of the wff's normal form (if one exists) — i.e. its normal form will have the same initial atom and the same number of arguments. Other results relating λ-calculus wffs in head-normal form to SKI-calculus wffs in lazy-normal form are stated and proved.

The ideas behind M. Schönfinkel's SKI-calculus, C.P. Wadsworth's λ-G-calculus, and D.A. Turner's SASL implementation are combined with the concept of lazy-normal form to produce a new deterministic combinator based graph and machine oriented reduction calculus: the LNF-calculus. The LNF-calculus is equivalent in power to the λ-calculus et al., but is much more directly and efficiently implementable. This is due primarily to the structure sharing properties of the LNF-calculus wffs. Both garbage nodes and forwarding arcs (indirection pointers), concepts that are usually relegated to a calculus' implementation, are given formal definitions in this calculus.

The design and experimental Lisp Machine implementation of LNF, a fully lazy higher order purely functional programming language with reduction semantics, are discussed. The LNF compiler transforms high level expressions into representations of LNF-calculus wffs. LNF's runtime system, a direct realization of the LNF-calculus' "is reducible to" relation, takes as input LNF-calculus wffs and produces irreducible wffs (wffs in lazy-normal form) as result. The thesis ends with brief discussions of alternate approaches to functional programming language compilation and runtime system organization.

Acknowledgements

I am greatly indebted to my advisor, Professor J. Alan Robinson, for his guidance and assistance throughout all stages of this research.

I am also very grateful to Professor F. Lockwood Morris, Heinz Schlütter, and Eric Boutteloup, who gave me valuable comments and suggestions on various drafts.

Furthermore I am thankful to my wife, Sue, for her unfailing support, endless patience, and encouragement.

Table of Contents

Chapter 1

Foundations

At its core, the implementation of LNF is a realization of a formal reduction calculus called the LNF-calculus. This chapter contains some preliminary conventions, definitions, and results concerning reduction calculi.

First, a formal definition of reduction calculi is given. Next, two reduction calculi: the λ-calculus ([Church 1936], [Church 1941]) and the SKI-calculus ([Schönfinkel 1924]) are presented. Schönfinkel's classic transformation of λ-calculus well-formed formulas (λ-wffs) into SKI-calculus well-formed formulas (SKI-wffs) is then defined. Wadsworth's concepts: head-redex and head-normal form are presented next. These concepts originally appeared in Wadsworth's Ph.D. thesis ([Wadsworth 1971]). Wadsworth's concept head-normal form is a generalization of Church's normal form in the λ-calculus (λ-normal form). The notions initial-redex and lazy-normal form are introduced. The notion lazy-normal form is a generalization of the SKI-calculus' normal form (SKI-normal form). One would not be far off by saying that lazy-normal form is to the SKI-calculus as head-normal form is to the λ-calculus. Some results relating the two calculi are then stated and proved. A few new results relating SKI-wffs in lazy-normal form to λ-wffs in head-normal form and to SKI-wffs in SKI-normal form are also proved. The chapter ends with a very brief discussion of λ-G-calculus [Wadsworth 1971] (a modification of the λ-calculus in which the wffs are rooted acyclic graphs).

1.1. Reduction Calculi

In the definitions to follow, the definienda appear in *italics*.

Definition 1.1: A *reduction calculus R* can be characterized by its set of well-formed formulas (*R-wff*) and a binary relation "immediately reducible to" (*R-imr*) on R-wff.

Reduction calculi, so defined, are exactly the "General Replacement Systems" of B.K. Rosen in [Rosen 1973].

BOLDFACE UPPERCASE identifiers will be used for meta-variables denoting arbitrary R-wffs. Different identifiers denote, in general, different R-wffs. The identity relation on R-wffs is denoted by "=".

Definition 1.2: Let R be a reduction calculus, defined by the set of well-formed formulas R-wff and binary relation R-imr.

- Let **A**, **B** ∈ R-wff. **A** *immediately reduces to* **B** iff the ordered pair <**A**,**B**> in R-imr. <**A**,**B**> in R-imr is often written **A** R-imr **B**.
- Let **A** ∈ R-wff. **A** is (*ir*)*reducible* iff there is (no) **B** in R-wff such that **A** R-imr **B**.
- The sequence $\mathbf{A}_1,\mathbf{A}_2, \ldots, \mathbf{A}_n$ is *a reduction sequence* of $\mathbf{A}_1$ iff $\mathbf{A}_i$ immediately reduces to $\mathbf{A}_{i+1}$, for $i = 1, \ldots, n-1$.
- *R-red* is the transitive closure of R-imr.
- Let **A**, **B** ∈ R-wff. **A** *reduces to* **B** (**B** is *a reduction of* **A**) iff **A** R-red **B**.
- *R-red** is the reflexive transitive closure of R-imr.
- Let **A**, **B**, **C** ∈ R-wff. If **A** R-red* **B** and **A** R-red* **C** implies the existence of a D in R-wff such that **B** R-red* **D** and **C** R-red* **D** then R is said to have the *Church-Rosser property* (R is Church-Rosser). The name comes from the work done by A. Church and J.B. Rosser in [Church 1936].
- R is *deterministic* iff R-imr is a partial function.

Note that any deterministic reduction calculus R trivially has the Church-Rosser property.

An Example of a Simple Reduction Calculus:

Definition 1.3: Let *SUM* be the reduction calculus defined by SUM-wff and SUM-imr.

Definition 1.4: *SUM-wff* is defined inductively as follows:

- Every integer is a SUM-wff.
- If **A** and **B** are SUM-wffs, then (**A** + **B**) is a SUM-wff.

Definition 1.5: *SUM-imr* is defined inductively as well:

- (**I** + **J**) SUM-imr **K**, for all integers **I**, **J**, and **K** where **K** is the sum of **I** and **J**.
- (**A** + **B**) SUM-imr (**C** + **B**), for all SUM-wffs **A**, **B**, and **C** where **A** SUM-imr **C**.
- (**A** + **B**) SUM-imr (**A** + **C**), for all SUM-wffs **A**, **B**, and **C** where **B** SUM-imr **C**.

From these definitions it can be seen that no integer is reducible ana that any SUM-wff **A** which is not an integer is reducible.

A reduction sequence (there are, of course, many others) of the SUM-wff (((3 + 2) + (0 + 10)) + (89 + 4)):

(((3 + 2) + (0 + 10)) + (89 + 4)),
(((3 + 2) + 10) + (89 + 4)),
((5 + 10) + (89 + 4)),
((5 + 10) + 93),
(15 + 93),
108.

It is easy to verify that even though SUM is not deterministic it is Church-Rosser.

1.2. The λ-calculus

The use of metavariables follows (for the most part) that of A. Church in [Church 1941]. **Boldface lowercase** identifiers denote variables. **BOLDFACE UPPERCASE** identifiers denote arbitrary λ-wffs.

Definition 1.6: Let *λ-calculus* be the reduction calculus defined by the set of well-formed formulas λ-wff and binary relation λ-imr.

1.2.1. Well-formed Formulas

It will often be convenient to use shorthand of the form FUNCTION-NAME[$\mathbf{ARG}_1, \ldots, \mathbf{ARG}_m$] to stand for λ-wffs and PREDICATE-NAME-P[$\mathbf{ARG}_1, \ldots, \mathbf{ARG}_n$] to stand for predications. For example, the piece of shorthand OPERATOR[A] (defined below) will stand in for a λ-wff and the shorthand VAR-P[A] (also defined below) will stand in for the predication "A is a variable". Before its use, each function and predicate will be given a formal definition. In these definitions, the author will make use of the following familiar forms:

- (*and* $\mathbf{C}_1 \cdots \mathbf{C}_n$)
- (*or* $\mathbf{D}_1 \cdots \mathbf{D}_m$)
- (*not* **B**)
- (*if* $\mathbf{B}_1$
   then $\mathbf{E}_1$
  [*elseif* $\mathbf{B}_i$
   then $\mathbf{E}_i$]*
  else $\mathbf{E}_n$)
- (*let* <NAME> *be* $\mathbf{E}_1$ *in* $\mathbf{E}_2$)
- ($\mathbf{E}_1$ *where* <NAME> *is* $\mathbf{E}_2$)

Definition 1.7: *VAR* is the set of all lowercase identifiers. Elements of VAR are called variables. Some examples of variables: "a", "flat", and "tire". For all variables **v**, *VAR-P*[**v**] is true.

Definition 1.8: *λ-wff* is defined inductively as follows:

- Every variable is a λ-wff.
- If **v** is a variable and **B** is a λ-wff, then (λ **v** **B**) is a λ-wff.
- If **A** and **B** are λ-wffs, then (**A** **B**) is a λ-wff.

Definition 1.9: Let **A** = (λ **v** **B**). **A** is an *abstraction* (*ABSTRACTION-P*[**A**]), **v** is the *bound variable* of **A** (**v** = *BV*[**A**]), and **B** is the *body* of **A** (**B** = *BODY*[**A**]).

Definition 1.10: Let **C** = (**A** **B**). **C** is a *combination* (*COMBINATION-P*[**C**]), **A** is the *operator* of **C** (**A** = *OPERATOR*[**C**]), and **B** is the *operand* of **C** (**B** = *OPERAND*[**C**]).

The pair of parentheses surrounding combinations is often omitted. Further, the combinations: ((**A** **B**) **C**), (((**A** **B**) **C**) **D**), etc. are written: **A** **B** **C**, **A** **B** **C** **D**, etc. Using this shorthand (association of combination to the left) for the combination ((**A** **B**) (**C** **D**)) results in **A** **B** (**C** **D**).

Definition 1.11: Let **A** ∈ λ-wff. The pair <**sf**,**B**>, where **sf** is a function (a composition of the selector functions BODY, OPERATOR, and OPERAND) and **B** is a λ-wff, is *a subformula* of **A** if **sf**[**A**] = **B**

Note that x is not a subformula of (λ x y). The phrases "**B** is a subformula of **A**", "**B** occurs in (the context of) **A**", and "**A** contains **B**" are often used in place of the somewhat unwieldy phrase "<**sf**,**B**> is a subformula of **A**".

Definition 1.12: Let **v** ∈ VAR and **B** ∈ λ-wff. The variable **v** *occurs free in* **B** (**v** *has a free occurrence in* **B**) iff

```
(or B = v
   (and B = (C D)
        v occurs free in either C or D)
   (and B = (λ u C)
        it is not the case that u = v
        v occurs free in C)).
```

Definition 1.13: Let **v** ∈ VAR and **B** ∈ λ-wff. The variable **v** *occurs bound in* **B** (**v** *has a bound occurrence in* **B**) iff

```
(or (and B = (λ u C)
         (or (and (u = v)
                  (u occurs in C))
             v occurs bound in C))
    (and B = (C D)
         v occurs bound in either C or D)).
```

It is possible that a variable has both free and bound occurrences in the same λ-wff. For example, consider the variable v in the λ-wff (v (λ v v)). Its occurrence in the operator is free and its occurrence in the operand is bound.

Definition 1.14: The *free* (*bound*) *variables* of a λ-wff **A** are those variables which have free (bound) occurrences in **A**.

Definition 1.15: Let **A** ∈ λ-wff. **A** is *closed* iff **A** has no free variables.

1.2.2. <u>Reduction</u>

The definition of the "immediately reducible to" relation in the λ-calculus depends directly on the notion of substitution. Informally, SUBST[**A**,**v**,**B**] (defined formally below) is **B** with all free occurrences of **v** in **B** replaced with **A**. Although it is easy to informally communicate the essence of the notion, it is also easy to make a mistake when writing out the formal definition. Besides having a complicated formalization, the function SUBST is expensive to implement. This is one of the reasons for basing the LNF-machine on the LNF-calculus — a reduction calculus without variables and substitution.

Definition 1.16: Let **v** ∈ VAR and **A**, **B** ∈ λ-wff.

SUBST[**A**,**v**,**B**] $\stackrel{\mathrm{Def}}{\rightarrow}$

```
(if B = v
  then A
 elseif VARP[B]
  then B
 elseif B = (C D)
  then (SUBST[A,v,C] SUBST[A,v,D])
 elseif B = (λ v C)
  then B
 elseif (and B = (λ u C)
             ;; it is not the case that u = v
             u does not occur free in A)
  then (λ u SUBST[A,v,C])
 else ;; B = (λ u C),
    ;; it is not the case that u = v, and
    ;; u occurs free in A)
   (λ x SUBST[A,v,SUBST[x,u,C]])
     where x is a variable which does not occur
         in either A or C).
```

Definition 1.17: (λ **v** **B**) *α-imc* (*α-converts*) (λ **u** SUBST[**u**,**v**,**B**]), for all λ-wffs **B** and all variables **u** and **v**, where **u** does not occur free in **B**.

Definition 1.18: ((λ **v** **B**) **A**) *β-imr* SUBST[**A**,**v**,**B**], for all variables **v** and λ-wffs **A** and **B**. Any λ-wff of the form ((λ **v** **B**) **A**) is a *β-redex* (*β-REDEX-P*[((λ **v** **B**) **A**)]). The λ-wff SUBST[**A**,**v**,**B**] is the *β-reductum* (*contraction*) of ((λ **v** **B**) **A**).

Definition 1.19: λ-*imr* is defined inductively:

- **A** λ-imr **B** if **A** α-imc **B**.
- **A** λ-imr **B** if **A** β-imr **B**.
- (**A** **B**) λ-imr (**C** **B**) if **A** λ-imr **C**.
- (**A** **B**) λ-imr (**A** **C**) if **B** λ-imr **C**.
- (λ **v** **B**) λ-imr (λ **v** **C**) if **B** λ-imr **C**.

The fiv? clauses in the definition of λ-imr are called *reduction rules* of the λ-calculus. The first two reduction rules differ from the other three. Both the first two rules "specify a redex-reductum pair" whereas the other three "specify a *reduction context* — i.e. a context in which a reduction may take place". For this reason the first two rules will be called *substantive reduction rules* and the others *contextual reduction rules*.

The contextual reduction rule:

(**A** **B**) λ-imr (**C** **B**) if **A** λ-imr **C**

says that the combination (**A** **B**) is a reduction context for **A**. Similarly, the contextual reduction rule:

(**A** **B**) λ-imr (**A** **C**) if **B** λ-imr **C**

states that the combination (**A** **B**) is a reduction context for **B**. Together these two rules indicate that the λ-calculus is nondeterministic. Anytime a single wff is a reduction context for more than one subformula, the "immediately reducible to" relation (if nonempty) will not be a partial function.

Definition 1.20: Let **A**, **B** be λ-wffs. In case **A** λ-imr **B** by virtue of the fact that **ASF** β-imr **BSF** where **ASF** (**BSF**) is a subformula of **A** (**B**), then **ASF** is the *redex contracted* in the reduction from **A** to **B**

Definition 1.21: λ-*red* is the transitive closure of the relation λ-imr.

Definition 1.22: λ-*red** (β-*red**) is the reflexive transitive closure of the relation λ-imr (β-imr).

Theorem 1.1: The λ-calculus is Church-Rosser. For the proof, see [Church 1941].

Definition 1.23: A λ-wff **E** is in λ-*normal form* (λ-*NF-P*[**E**]) iff **E** does not contain any β-redexes.

Definition 1.24: Let **A**, **B** be λ-wffs. If **A** λ-red* **B** and λ-NF-P[**B**], then **B** is *a* λ-*normal form of* **A**.

Definition 1.25: Let **A** ∈ λ-wff. Assume (*not* λ-NF-P[**A**]). By definition, **A** contains at least one β-redex. The *leftmost occurrence of a β-redex* of **A**, (*LEFTMOST-β-REDEX*[**A**]) is defined as follows.

```
LEFTMOST-β-REDEX[A] $\overset{\text{Def}}{=}$
  (if β-REDEX-P[A]
    then A
   elseif A = (λ v B)
    then LEFTMOST-β-REDEX[B]
   else ;; A = (B C)
    (if B contains a β-redex
      then LEFTMOST-β-REDEX[B]
     else LEFTMOST-β-REDEX[C]))
```

Definition 1.26: Let **A**, **B** be λ-wffs. **A** λ-*normal-imr* **B** iff **A** λ-imr **B** and the redex contracted was LEFTMOST-β-REDEX[**A**].

In [Church 1941], the reduction calculus λ-calculus is called the "calculus of λ-K-conversion", the relation λ-red* is called "conv-I-II", and the relation λ-normal-imr is called a "reduction of order one".

Church's "calculus of λ-conversion" (also presented in [Church 1941]) differs from his "calculus of λ-K-conversion" in the definition of well-formed formulas. In Church's λ-conversion calculus, an expression of the form (λ v B) is well-formed only if there is at least one free occurrence of v in B.

Definition 1.27: λ-*normal-red** is the reflexive transitive closure of λ-normal-imr.

Definition 1.28: Let **A**, **B** be λ-wffs. **B** is *a* λ-*normal reduction of* **A** iff **A** λ-normal-red* **B**.

Definition 1.29: Let $\mathbf{A}_1, \mathbf{A}_2, \ldots, \mathbf{A}_n$ be λ-wffs. $\mathbf{A}_1, \mathbf{A}_2, \ldots, \mathbf{A}_n$ is a *λ-normal order reduction sequence* iff $\mathbf{A}_i$ λ-normal-imr $\mathbf{A}_{i+1}$, $i = 1, \ldots, n-1$.

Theorem 1.2: *The λ-NF Standardization Theorem.* Let **A** ∈ λ-wff. **A** has a λ-normal form iff there exists a λ-wff **B** such that λ-NF-P[**B**] and **A** λ-normal-red* **B**. For the proof, see [Church 1941].

Note that the reduction calculus characterized by the sets λ-wff and λ-normal-imr is a deterministic one. This is true because each λ-wff contains at most one leftmost β-redex and, hence, is a reduction context for at most one of its subformulas.

1.2.3. Head-normal Form

Head-normal form is a generalization of the concept of λ-normal form — i.e. a λ-wff may have a head-normal form without having a λ-normal form.

Definition 1.30: Let **A** ∈ λ-wff. **A** *contains a head-redex* **R** iff

```
(or β-REDEX-P[A]
    (and A = (λ v B)
         B contains a head-redex)
    (and A = (B C)
         (not β-REDEX-P[A])
         B contains a head-redex)).
```

Definition 1.31: Let **A** ∈ λ-wff contain a head-redex. The *head-redex of* **A** is defined to be *HEAD-REDEX*[**A**] where:

```
HEAD-REDEX[A] $\overset{\text{Def}}{\rightarrow}$
  (if β-REDEX-P[A]
    then A
   elseif A = (λ v B)
    then HEAD-REDEX[B]
   else HEAD-REDEX[OPERATOR[A])
```

Definition 1.32: Let **A** ∈ λ-wff. **A** is *in head-normal form* (*HEAD-NF-P*[**A**]) iff

```
(or VAR-P[A]
    (and A = (λ v B)
         HEAD-NF-P[B])
    (and A = (B C)
         (not β-REDEX-P[A])
         HEAD-NF-P[B])).
```

Some notes on head-normal form:

- An alternate definition for a λ-wff **A** being in head-normal form is that **A** is in head-normal form iff **A** does not contain a head-redex.
- A λ-wff in head-normal form always looks like: $(\lambda \mathbf{x}_1 \cdots (\lambda \mathbf{x}_n (\mathbf{v}\ \mathbf{B}_1 \cdots \mathbf{B}_m) \cdots))$, $n, m \geq 0$
- A λ-wff not in head-normal form always look like: $(\lambda \mathbf{x}_1 \cdots (\lambda \mathbf{x}_n (((\lambda \mathbf{v}\ \mathbf{B})\ \mathbf{A})\ \mathbf{B}_1 \cdots \mathbf{B}_m) \cdots))$, $n, m \geq 0$.

Definition 1.33: Let **A**, **B** be λ-wffs. **A** *head-imr* **B** iff **A** λ-imr **B** and the redex contracted is the HEAD-REDEX[**A**].

Definition 1.34: *head-red** is the reflexive transitive closure of head-imr.

Definition 1.35: Let **A**, **B** be λ-wffs. **B** is *a head reduction of* **A** iff **A** head-red* **B**.

As mentioned above, the concepts head-normal form, head-redex, and head reduction (defined above) appeared originally in [Wadsworth 1971].

Theorem 1.3: Let **A** ∈ λ-wff If **A** has a λ-normal form, then **A** has a head-normal form. However, **A** having a head-normal form does not imply that **A** has a λ-normal form. The λ-wff (v ((λ x (x x)) (λ x (x x)))) is an example of a λ-wff which has a head-normal form (it is in head-normal form) but has no λ-normal form.

The theorem above says, simply, that the subset of λ-wffs having a head-normal form contains the subset of λ-wffs having a normal form.

Theorem 1.4: *The HEAD-NF Standardization Theorem.* Let **A** ∈ λ-wff. **A** has a head-normal form iff there exists a λ-wff **B** such that HEAD-NF-P[**B**] and **A** head-red* **B**. For the proof, see [Wadsworth 1971].

The reduction calculus characterized by the sets λ-wff and head-imr, like the calculus based on the sets λ-wff and λ-normal-imr, is deterministic.

1.3. The SKI-calculus

The SKI-calculus, as presented herein, is essentially Schönfinkel's Funktionenkalkül (with Schönfinkel's functor C renamed to K) presented in [Schönfinkel 1924]. The SKI-calculus is equivalent in power to the λ-calculus.

Definition 1.36: Let the reduction calculus *SKI-calculus* be defined by the set of well-formed formulas SKI-wff and the binary relation SKI-imr.

1.3.1. Well-formed Formulas

Definition 1.37: *SKI-wff* is defined inductively as follows:

- Every variable is an SKI-wff.
- The *functors* S, K, and I are SKI-wffs. For all functors **X**, *FUNCTOR-P*[**X**]. These functors are also called *combinators*.
- For all SKI-wffs **A** and **B**, the combination (**A B**) is an SKI-wff.

Definition 1.38: An *atom* is either a variable or one of the functors S, K, or I. For all atoms **X**, *ATOM-P*[**X**].

Boldface lowercase identifiers now stand for arbitrary atoms, not just variables. **BOLDFACE UPPERCASE** identifiers now stand for arbitrary SKI-wffs.

Definition 1.39: Note that every SKI-wff can be written in the form:

$$\mathbf{a}\ \mathbf{E}_1 \cdots \mathbf{E}_{n\,,}\ n \geq 0.$$

The atom **a** is the *initial atom* of the SKI-wff. The SKI-wffs $\mathbf{E}_1, \ldots, \mathbf{E}_n$ are the *arguments* of the SKI-wff and $\mathbf{E}_i$ is the SKI-wff's *ith argument*.

Definition 1.40: Let **A** ∈ SKI-wff. The pair <**sf**,**B**>, where **sf** is a function (a composition of the selector functions OPERATOR and OPERAND) and **B** is a SKI-wff, is *a subformula* of **A** if **sf**[**A**] = **B**.

Definition 1.41: Let **X** ∈ SKI-wff have the two subformulas: <**yf**,**Y**> and <**zf**,**Z**>. These subformulas are *disjoint* iff there is no function **f** such that **yf** = **f**∘**zf** (where ∘ denotes functional composition) in which case **Z** contains **Y**, or **zf** = **f**∘**yf** in which case **Y** contains **Z**.

1.3.2. Reduction

Reduction in the SKI-calculus does not depend on the notion of substitution. Thus, the relation SKI-imr is much easier to formalize than λ-imr.

Definition 1.42: *SKI-imr* is defined inductively:

- S **F G X** SKI-imr **F X** (**G X**).
- K **X Y** SKI-imr **X**.
- I **X** SKI-imr **X**.
- **A B** SKI-imr **C B** if **A** SKI-imr **C**.
- **A B** SKI-imr **A C** if **B** SKI-imr **C**.

The five clauses in the above definition of SKI-imr are called the *reduction rules* of the SKI-calculus. The first three are the calculus' *substantive reduction rules* and the other two its *contextual reduction rules*. It is easy to see that the SKI-calculus, like the λ-calculus, is nondeterministic.

Definition 1.43: An SKI-wff **E** is an *SKI-redex* iff (*SKI-REDEX-P*[E]) where
SKI-REDEX-P[E] $\overset{\text{Def}}{\Rightarrow}$
(*or* **E** = S **F G X**
E = K **X Y**
E = I **X**).

Definition 1.44: (from [Sanchis 1967]) Let **X** ∈ SKI-wff. Let **U** and **Z** be SKI-redexes contained in **X**. Let **Y** be the SKI-wff which results from contracting **U** — i.e. **X** SKI-imr **Y**. The *residuals* of **Z** in **Y** are as follows (each of the residuals will be an occurrence of an SKI-redex in **Y**):

- If **Z** is **U**, then there are no residuals.
- If **Z** is disjoint from **U**, then (since **Z** is unaffected by the contraction) the corresponding occurrence of **Z** in **Y** is the residual of **Z**.
- If **Z** is contained in **U**, then (depending on which type of SKI-redex **U** is) there are zero, one or two residuals. There are none in case **U** = K **A B** and **Z** is in **B**. There is one in case **U** = I **A**, K **A B**, S **A B C**, or S **B A C** and **Z** is in **A** — it is the occurrence of **Z** in **A** which is in **Y**. There are two in case **U** = S **A B C** and **Z** is in **C** — each of the occurrences of **Z** in the two occurrences of **C** which are in **Y**.
- If **Z** contains U, then the residual is $\mathbf{Z}_1$, where $\mathbf{Z}_1$ is the SKI-wff in **Y** such that **Z** SKI-imr $\mathbf{Z}_1$ by virtue of contracting U.

Observe that every residual of **Z** is an occurrence of an SKI-redex having the same initial atom and same number of arguments — i.e. the same type of SKI-redex as **Z**.

Definition 1.45: *SKI-red* is the transitive closure of SKI-imr.

Definition 1.46: *SKI-red** is the reflexive transitive closure of SKI-imr.

Lemma 1.1: Let **X** ∈ SKI-wff. If **X** SKI-imr $\mathbf{Y}_1$ (by virtue of contracting SKI-redex U_1) and **X** SKI-imr $\mathbf{Y}_2$ (by virtue of contracting SKI-redex $\mathbf{U}_2$), then there is an SKI-wff **Z** such that $\mathbf{Y}_1$ SKI-red* **Z** (by virtue of contracting the residuals of SKI-redex U_2) and $\mathbf{Y}_2$ SKI-red* **Z** (by virtue of contracting the residuals of SKI-redex U_1). For a proof, see [Sanchis 1967].

Theorem 1.5: The SKI-calculus is Church-Rosser. For a proof, see [Sanchis 1967].

Definition 1.47: An SKI-wff E is *in SKI-normal form* (*SKI-NF-P*[E]) iff it does not contain any SKI-redexes.

Definition 1.48: Let E, F ∈ SKI-wff. F is *the SKI-normal form of* E iff SKI-NF-P[F] and E SKI-red* F.

Definition 1.49: Let E, F ∈ SKI-wff. E *is equivalent to* F (EQUIVALENT-P[E,F]) if the SKI-normal form of **E** = the SKI-normal form of **F**.

Informally, two equivalent SKI-wffs are said to be different representations of the same object. The SKI-normal form is thought of as the preferred (or canonical) representation of the object.

Definition 1.50: Let **A** ∈ SKI-wff. Assume (*not* SKI-NF-P[**A**]). By definition, **A** contains at least one SKI-redex. The *leftmost occurrence of an SKI-redex in* **A** is (*LEFTMOST-SKI-REDEX*[**A**]) where

```
LEFTMOST-SKI-REDEX[A]) $\overset{\text{Def}}{\rightarrow}$
  (if SKI-REDEX-P[A]
     then A
   elseif OPERATOR[A] contains an SKI-redex
     then LEFTMOST-SKI-REDEX[OPERATOR[A]]
   else LEFTMOST-SKI-REDEX[OPERAND[A]])
```

Definition 1.51: Let **A**, **B** ∈ SKI-wff **A** *SKI-normal-imr* **B** iff **A** SKI-imr **B** and the redex contracted was the LEFTMOST-SKI-REDEX[**A**]

Definition 1.52: Let $\mathbf{A}_1$,$\mathbf{A}_2$, . . . $\mathbf{A}_n$ ∈ SKI-wff. $\mathbf{A}_1$,$\mathbf{A}_2$, . . . , $\mathbf{A}_n$ is *an SKI-normal order reduction sequence* iff $\mathbf{A}_i$ SKI-normal-imr $\mathbf{A}_{i+1}$, $i=1$, . . . , $n-1$.

Definition 1.53: *SKI-normal-red** is the reflexive transitive closure of SKI-normal-imr.

Definition 1.54: Let **A**, **B** ∈ SKI-wff. **B** is *an SKI-normal reduction of* **A** iff **A** SKI-normal-red* **B**.

Theorem 1.6: *The SKI-NF Standardization Theorem.* Let **A** ∈ SKI-wff. **A** has an SKI-normal form iff there exists an SKI-wff **B** such that SKI-NF-P[**B**] and **A** SKI-normal-red* **B**. For the proof, see [Curry 1958].

The reduction calculus determined by the sets SKI-wff and SKI-normal-imr is deterministic, since the leftmost SKI-redex (if it exists) is unique.

1.3.3. Lazy-normal Form

It was stated in the introduction to this chapter that the concept lazy-normal form in the SKI-calculus is not unlike the concept of head-normal form in the λ-calculus. Lazy-normal form is defined in this section and then later on it is shown that λ-wffs in head-normal form, when transformed into SKI-wffs via Schönfinkel's abstraction algorithm, are in lazy-normal form.

Definition 1.55: Let **E** ∈ SKI-wff. **E** *contains an initial redex* iff

```
(or SKI-REDEX-P[E]
    (and COMBINATION-P[E]
         OPERATOR[E] contains an initial redex))
```

Definition 1.56: Let **E** ∈ SKI-wff contain an initial redex. The *initial redex of* **E** is defined to be (*INITIAL-REDEX*[**E**]) where

```
INITIAL-REDEX)[E] $\overset{\text{Def}}{\rightarrow}$
  (if SKI-REDEX-P[E]
     then E
   else INITIAL-REDEX[OPERATOR[A]])
```

The SKI-redexes which are not initial redexes are called *internal redexes* since for an SKI-wff $\mathbf{X} = \mathbf{a}\,\mathbf{X}_1 \cdots \mathbf{X}_n$, each of its internal redexes are contained in one its $\mathbf{X}_i$ s.

Definition 1.57: An SKI-wff **E** is *in lazy-normal form* (*LAZY-NF-P*[**E**]) iff **E** does not contain an initial redex.

Observe, therefore, LAZY-NF-P[**E**] iff **E** is an atom, or **E** is a combination but not an SKI-redex, and the operator of **E** is in lazy-normal form.

Definition 1.58: Let **E**, **F** ∈ SKI-wff. **F** is *a lazy-normal form of* **E** iff LAZY-NF-P[**F**] and **E** SKI-red* **F**.

Definition 1.59: Let **A**, **B** ∈ SKI-wff. **A** *lazy-imr* **B** iff **A** SKI-imr **B** and the redex contracted was the INITIAL-REDEX[**A**].

Definition 1.60: Let **A**, **B** ∈ SKI-wff. **A** *internal-imr* **B** iff **A** SKI-imr **B** and the redex contracted was an internal redex.

It may be noted that the reduction calculus characterized by the set of well formed formulas SKI-wff and the relation lazy-imr is deterministic.

Definition 1.61: The relation *lazy-red** (*internal-red**) is the reflexive transitive closure of lazy-imr (internal-imr).

Some observations concerning initial and internal redexes:

- An SKI-wff contains at most one initial redex.
- If an SKI-wff contains an initial redex **X** then **X** is also the SKI-wff's leftmost SKI-redex.
- An SKI-wff not in SKI-normal form always contains a leftmost SKI-redex but need not contain an initial redex. For example, consider the SKI-wff **X** = (K (I I)). **X**'s leftmost SKI-redex is (I I) but **X** does not contain an initial redex.
- If **X** internal-imr **Y**, then **Y** has the same initial atom and the same number of arguments as does **X**. It then follows that **Y** contains an initial redex **IR′** iff **X** contains an initial redex **IR** and **IR′** is the residual of **IR** in **Y**.

Lemma 1.2: Let **X** ∈ SKI-wff. If **X** internal-red* **Y** and **Y** lazy-imr **Z**, then there is a **W** such that **X** lazy-red* **W** and **W** internal-red* **Z**

Proof:

It has been noted that if **A** internal-imr **B** and **B** lazy-imr **C**,
then the initial redex in **B** is the residual of an initial redex in **A**.
Let **IRY** be the initial redex contained in **Y**
It follows from the preceding remark that **X** contains an initial redex (call it **IRX**)
and that **IRY** is the residual of the residual of the residual of . . of **IRX**
It may also be observed that **IRX** and **IRY** have the same initial atom
and the same number of arguments -- they are the same type of SKI-redex.
Let $\mathbf{X} = \mathbf{a}\,\mathbf{X}_1 \cdots \mathbf{X}_m$.
X internal-red* **Y** implies $\mathbf{Y} = \mathbf{a}\,\mathbf{Y}_1 \cdots \mathbf{Y}_m$ and $\mathbf{X}_i$ SKI-red* $\mathbf{Y}_i$
IRX must be $\mathbf{a}\,\mathbf{X}_1 \cdots \mathbf{X}_k$ and **IRY** must be $\mathbf{a}\,\mathbf{Y}_1 \cdots \mathbf{Y}_k$ for some $k = 1, 2, or\ 3$
By repeated application of Lemma 1.1, it follows that
there is an **X′** such that **X** lazy-imr **X′** and **X′** SKI-red* **Z**.

where the redexes contracted from **X**$'$ to **Z** are residuals of
the redexes contracted from **X** to **Y**.

This **X**$'$ is either:

$\mathbf{X}_1 \cdots \mathbf{X}_m$ (in case **a** = I), or

$\mathbf{X}_1\, \mathbf{X}_3 \cdots \mathbf{X}_m$ (in case **a** = K), or

$\mathbf{X}_1\, \mathbf{X}_3\, (\mathbf{X}_2\, \mathbf{X}_3)\, \mathbf{X}_4 \cdots \mathbf{X}_m$ (in case **a** = S).

Therefore, the only initial redexes that could be contracted in the reduction from **X**$'$ to **Z** must be residuals of initial redexes contracted in the reduction from $\mathbf{X}_1$ to $\mathbf{Y}_1$.

It suffices to show that there exists a $\mathbf{W}_1$ such that $\mathbf{X}_1$ lazy-red* $\mathbf{W}_1$ and $\mathbf{W}_1$ internal-red* $\mathbf{Y}_1$.

If $\mathbf{X}_1$ internal-red* $\mathbf{Y}_1$, then done.

So, suppose there is $\mathbf{X}_1{}'$ and $\mathbf{X}_1{}''$ such that
$\mathbf{X}_1$ internal-red* $\mathbf{X}_1{}'$ lazy-imr $\mathbf{X}_1{}''$ SKI-red* $\mathbf{Y}_1$.

This situation is similar to the original problem.

There is an important difference, however.

The initial redex contracted in the reduction from $\mathbf{X}_1{}'$ to $\mathbf{X}_1{}''$ is the residual of a redex *strictly contained* in IRX.

Therefore, the argument up to this point may be repeated with $\mathbf{X}_1$ as **X**, $\mathbf{X}_1{}'$ as **Y**, and $\mathbf{X}_1{}''$ as **Z**.

Since all SKI-wffs are finite, eventually there will be a first argument of the initial redex which does not strictly contain an initial redex.

End Proof

Lemma 1.3: Let **X** ∈ SKI-wff. If **X** SKI-red* **Z**, then there is an SKI-wff **Y** such that **X** lazy-red* **Y** and **Y** internal-red* **Z**.

Proof:

Proof is by induction on the length of the SKI-reduction sequence from **X** to **Z**.

Case 1: n=0 and n=1. Trivial.

Case 2: Lemma holds for reduction sequences of length equal to n.

To show: Lemma holds for reductions of length *n+1*.

Let the reduction sequence from **X** to **Z** be:

$\mathbf{X}_0, \ldots, \mathbf{X}_n, \mathbf{X}_{n+1}$ where $\mathbf{X} = \mathbf{X}_0$ and $\mathbf{Z} = \mathbf{X}_{n+1}$.

By the induction hypothesis, there is an SKI-wff **W** such that
$\mathbf{X}_0$ lazy-red* **W** and **W** internal-red* $\mathbf{X}_n$.

If $\mathbf{X}_n$ internal-imr $\mathbf{X}_{n+1}$, then let **Y** be **W**. Done.

So, suppose $\mathbf{X}_n$ lazy-imr $\mathbf{X}_{n+1}$

It is also the case that **W** internal-red* $\mathbf{X}_n$.

By Lemma 1.2, there exists a **Y** such that **W** lazy-red* **Y** and **Y** internal-red* $\mathbf{X}_{n+1}$.

Therefore, since $\mathbf{X}_0$ lazy-red* **W**, $\mathbf{X}_0$ lazy-red* **Y** and **Y** internal-red* $\mathbf{X}_{n+1}$.

End Proof

Theorem 1.7: Let **A** ∈ SKI-wff. **A** has a lazy-normal form iff there exists an SKI-wff **B** such that LAZY-NF-P[**B**] and **A** lazy-red* **B**.

Proof:

⇐) Trivial. **B** is a lazy-normal form of **A**.

⇒) Let **C** be a lazy-normal form of **A**.

This implies LAZY-NF-P[**C**] and **A** SKI-red* **C**.

By Lemma 1.3, there is a **B** such that **A** lazy-red* **B** and **B** internal-red* **C**.

LAZY-NF-P[**B**] since if **B** contains an initial redex then

C would contain an initial redex.

End Proof

Theorem 1.8: Let $\mathbf{X} \in$ SKI-wff. If **Y** and **Z** are lazy normal forms of **X** and $\mathbf{Y} = \mathbf{a}\, \mathbf{Y}_1 \cdots \mathbf{Y}_n$ for some $n \geq 0$, then $\mathbf{Z} = \mathbf{a}\, \mathbf{Z}_1 \cdots \mathbf{Z}_n$ and there is an SKI-wff $\mathbf{W} = \mathbf{a}\, \mathbf{W}_1 \cdots \mathbf{W}_n$ such that $\mathbf{Y}_i$ SKI-red* $\mathbf{W}_i$ and $\mathbf{Z}_i$ SKI-red* $\mathbf{W}_i$, $1 \leq i \leq n$. A Church-Rosser like property.

Proof:

By Lemma 1.3, there is a **U** such that **X** lazy-red* **U**, **U** internal-red* **Y**, and **U** in lazy-normal form.

$\mathbf{Y} = \mathbf{a}\, \mathbf{Y}_1 \cdots \mathbf{Y}_n$ implies, since internal reduction sequences do not change either the initial atom or the number of arguments, **U** must be of the form: $\mathbf{a}\, \mathbf{U}_1 \cdots \mathbf{U}_n$ and $\mathbf{U}_i$ SKI-red* $\mathbf{Y}_i$, $i = 1, \ldots, n$.

Similary, there is a **V** such that **X** lazy-red* **V**, **V** internal-red* **Z**, and **V** in lazy-normal form.

Since both **U** and **V** are lazy-reductions from **X**, initial redexes are unique, and both **U** and **V** are in lazy-normal form, it must be the case that $\mathbf{U} = \mathbf{V}$.

Thus, $\mathbf{V} = \mathbf{a}\, \mathbf{U}_1 \cdots \mathbf{U}_n$, $\mathbf{Z} = \mathbf{a}\, \mathbf{Z}_1 \cdots \mathbf{Z}_n$, and $\mathbf{U}_i$ SKI-red* $\mathbf{Z}_i$,$1 \leq i \leq n$.

Since $\mathbf{U}_i$ SKI-red* $\mathbf{Y}_i$ and $\mathbf{U}_i$ SKI-red* $\mathbf{Z}_i$, by the Church-Rosser property, there is a $\mathbf{W}_i$ such that $\mathbf{Y}_i$ SKI-red* $\mathbf{W}_i$ and $\mathbf{Z}_i$ SKI-red* $\mathbf{W}_i$,$1 \leq i \leq n$.

Let $\mathbf{W} = \mathbf{a}\, \mathbf{W}_1 \cdots \mathbf{W}_n$.

End Proof

Theorem 1.9: Let $\mathbf{E} \in$ SKI-wff. If **E** has an SKI-normal form, then **E** has a lazy-normal form. However. **E** having a lazy-normal form does not imply that **E** has an SKI-normal form.

The proof of the above theorem is trivial The SKI-wff S ((S I I) (S I I)) is an example of an SKI-wff which has a lazy-normal form (it is in lazy-normal form) but has no SKI-normal form.

Theorem 1.10: Let $\mathbf{A} \in$ SKI-wff. If **A** has an SKI-normal form **B**, then there is an SKI-wff **C** such that **A** lazy-red* **C** (**C** in lazy-normal form) and **C** SKI-normal-red* **B**.

Proof:

By Theorem 1.6, there is an SKI-normal order reduction sequence $\mathbf{A}_1, \ldots, \mathbf{A}_n$ where $\mathbf{A}_1 = \mathbf{A}$ and $\mathbf{A}_n = \mathbf{B}$.

Either $\mathbf{A}_1$ is in lazy-normal form or its not. If it is, then the proof is complete.

Suppose, therefore, that $\mathbf{A}_1$ not in lazy-normal form.

By the definition of lazy-normal form, $\mathbf{A}_1$ contains an initial redex.

It has been observed that initial redexes are also leftmost SKI-redexes.

Thus, the redex contracted in the reduction from $\mathbf{A}_1$ to $\mathbf{A}_2$ is $\mathbf{A}_1$'s initial redex implying that $\mathbf{A}_1$ lazy-imr $\mathbf{A}_2$.

This same argument may be applied to the SKI-wffs $\mathbf{A}_2, \ldots, \mathbf{A}_{n-1}$.

There are two cases to consider.

Either it is found that one of these SKI-wffs is in lazy-normal form or that none of them are in lazy-normal form.

Suppose at least one of them is in lazy-normal form.

Let $\mathbf{A}_j$ be the one having the smallest index.

By the preceding argument, $\mathbf{A}_1$ lazy-red* $\mathbf{A}_j$ and the proof is complete.

Otherwise, $\mathbf{A}_1$ lazy-red* $\mathbf{A}_n$

Since $\mathbf{A}_n$ is in SKI-normal form it is also in lazy-normal form.

End Proof

1.4. Relating the λ-calculus and the SKI-calculus

Definition 1.62: Let **E** ∈ λ-wff. The *SKI-transform* of **E** is the SKI-wff λ-*TO-SKI*[**E**] where

```
λ-TO-SKI[E] $\overset{\mathrm{Def}}{\rightarrow}$
  (if VAR-P[E]
    then E
   elseif E = (λ v B)
    then ABSTRACT[v,λ-TO-SKI[B]]
   else (λ-TO-SKI[OPERATOR[E]] λ-TO-SKI[OPERAND[E]]))
```

Definition 1.63: For any variable **v** and SKI-wff **B**, there is an SKI-wff *ABSTRACT*[**v**,**B**] where

```
ABSTRACT[v,B] $\overset{\mathrm{Def}}{\rightarrow}$
  (if B = v
    then I
   elseif v does not occur in B
    then K B
   else
    S ABSTRACT[v,OPERATOR[B]] ABSTRACT[v,OPERAND[B]])
```

The transformation of expressions containing bound variables into expressions without bound variables (called ABSTRACTion) was first presented in [Schönfinkel 1924]. It was Schönfinkel's aim "to make the number of undefined notions as small as we can". In the case of the transformation from λ-wffs to SKI-wffs, the arbitrary abstractions present in the λ-calculus have been replaced with the three special functors (abstractions): S, K, and I.[1]

The SKI-wff λ-TO-SKI[**EXP**] is similar to Church's "the combination belonging to **EXP**". In [Church 1941], the transformed expressions were well-formed formulas of "the calculus of λ-conversion". That set of well-formed formulas, as mentioned above, did not contain abstractions having no free occurrences of the bound variable in the body. λ-TO-SKI[**EXP**] is called "the H-transform of **EXP**" in [Hindley 1972].

Hindley et al. also define, for SKI-wffs **EXP**, "the λ-transform of **EXP**". Herein the λ-transform of an SKI-wff **EXP** will be denoted by the λ-wff SKI-TO-λ[**EXP**] defined below.

[1] In the same paper, Schonfinkel showed that the functor I was unnecessary as it could be represented by S and K with the SKI-wff S K K He even went on to demonstrate that the functors S and K could be defined in terms of a single functor he called J These representation tricks, however, are not as remarkable as his "bound variable eliminating" transformation just defined

Definition 1.64: Let **EXP** ∈ SKI-wff. The *λ-transform* of **EXP** is *SKI-TO-λ*[**EXP**] where

```
SKI-TO-λ[EXP] ≝
 (if VAR-P[EXP]
   then EXP
  elseif EXP = I
   then (λ x x)
  elseif EXP = K
   then (λ x (λ y x))
  elseif EXP = S
   then (λ f (λ g (λ x (f x (g x)))))
  else ;; it is a combination
   (SKI-TO-λ[OPERATOR[EXP]] SKI-TO-λ[OPERAND[EXP]]))
```

1.4.1. Some Results

Some simple results which relate λ-wffs to SKI-wffs are stated and proved below.

Lemma 1.4: *Redex Preservation Lemma.* Let **EXP** ∈ λ-wff. If **EXP′** = λ-TO-SKI[**EXP**], then β-REDEX-P[**EXP**] iff SKI-REDEX-P[**EXP′**].

Proof:

First suppose β-REDEX-P[**EXP**]. To show: SKI-REDEX-P[**EXP′**].

β-REDEX-P[**EXP**] implies **EXP** = ((λ **v B**) **A**) for some variable **v** and λ-wffs **A** and B

By the definition of λ-TO-SKI,

EXP′ = (**RATOR′ RAND′**), where **RATOR′** = ABSTRACT[**v**,**B′**], where **B′** = λ-TO-SKI[**B**]) and **RAND′** = λ-TO-SKI[**A**].

There are two cases to consider. **B′** is either an atom or a combination.

Case 1: ATOM-P[**B′**].

B′ is either **v** or it is not.

Case 1a: **v** = **B′**.

By definition of ABSTRACT, **RATOR′** = I.

RATOR′ = I implies **EXP′** = I **RAND′** which implies SKI-REDEX-P[**EXP′**].

Case 1b: It is not the case that **v** = **B′**.

By definition of ABSTRACT, **RATOR′** = (K **B′**).

RATOR′ = (K **B′**) implies **EXP′** = K **B′ RAND′**, which implies SKI-REDEX-P[**EXP′**].

Case 2: COMBINATION-P[**B′**].

Either **v** occurs in **B′** or it doesn't.

Case 2a: **v** occurs in **B′**.

By definition of ABSTRACT, **RATOR′** = S **RT′ RN′**.

RATOR′ = S **RT′ RN′** implies **EXP′** = S **RT′ RN′ RAND′**, which implies SKI-REDEX-P[**EXP′**].

Case 2b: **v** does not occur in **B′**.

By definition of ABSTRACT, **RATOR′** = (K **B′**).

Same as case 1b.

Hence, if β-REDEX-P[**EXP**], then SKI-REDEX-P[**EXP′**].

Now suppose SKI-REDEX-P[**EXP′**]. To show: β-REDEX-P[**EXP**].

SKI-REDEX-P[**EXP′**] implies COMBINATION-P[**EXP′**].

Let **EXP′** = (**RATOR′ RAND′**).
By the definition of λ-TO-SKI, COMBINATION-P[**EXP**].
Let **EXP** = (**RATOR RAND**).
The definition of λ-TO-SKI also implies
RATOR′ = λ-TO-SKI[**RATOR**] and **RAND′** = λ-TO-SKI[**RAND**].
SKI-REDEX-P[**EXP′**] implies **EXP′** has one of three forms:
Case 1: **EXP′** = I **X**, for some SKI-wff **X**.
By definition of **EXP′**, **RATOR′** = I.
If **RATOR′** = I, then **RATOR** = (λ v v), for some variable v
which implies β-REDEX-P[**EXP**].
Case 2: **EXP′** = K **X Y**, for some SKI-wffs **X** and **Y**.
By definition of **EXP′**, **RATOR′** = K **X**.
If **RATOR′** = K **X**, then **RATOR** = (λ v **A**),
for some variable v and λ-wff **A**. This implies β-REDEX-P[**EXP**].
Case 3: **EXP′** = S **F G X**, for some SKI-wffs **F**, **G**, and **X**.
By definition of **EXP′**, **RATOR′** = S **F G**.
If **RATOR′** = S **F G**, then **RATOR** = (λ v **A**), for some variable v and λ-wff **A**.
This implies β-REDEX-P[**EXP**].
Hence, if SKI-REDEX-P[**EXP′**], then β-REDEX-P[**EXP**].
Therefore, β-REDEX-P[**EXP**] iff SKI-REDEX-P[**EXP′**].

End Proof

Theorem 1.11: *ABSTRACTion preserves SKI-normal form.* Let v ∈ VAR and **BODY** ∈ SKI-wff. If **EXP** = ABSTRACT[v,**BODY**] and SKI-NF-P[**BODY**], then SKI-NF-P[**EXP**].

Proof:

Proof is by structural induction on **BODY**. There are two cases to consider:
BODY is either an atom or a combination.
Case 1: ATOM-P[**BODY**]. There are two sub-cases to consider:
Case 1a: v = **BODY**.
By definition of ABSTRACT, **EXP** = I. ATOM-P[**EXP**] implies SKI-NF-P[**EXP**].
Case 1b: It is not the case that v = **BODY**.
By definition of ABSTRACT, **EXP** = (K **BODY**).
By definition of SKI-REDEX-P, (*not* SKI-REDEX-P[**EXP**]).
This and the facts: SKI-NF-P[K] and (by hypothesis) SKI-NF-P[**BODY**] ,
imply SKI-NF-P[**EXP**].
Case 2: **BODY** = **RATOR RAND**. There are two sub-cases to consider:
Case 2a: v occurs in **BODY**.
By definition of ABSTRACT, **EXP** = (S **RATOR′ RAND′**), where
RATOR′ = ABSTRACT[v,**RATOR**] and **RAND′** = ABSTRACT[v,**RAND**].
By definition of SKI-REDEX-P, (*not* SKI-REDEX-P[**EXP**]) and
(*not* SKI-REDEX-P[(S **RATOR′**)].
By definition of SKI-NF-P, SKI-NF-P[**RATOR**] and SKI-NF-P[**RAND**].
By induction, SKI-NF-P[**RATOR′**] and SKI-NF-P[**RAND′**].
These facts together imply SKI-NF-P[**EXP**].
Case 2b: v does not occur in **BODY**.
By definition of ABSTRACT, **EXP** = (K **BODY**).
Same as case 1b.

End Proof

Let **v** ∈ VAR and **BODY** ∈ SKI-wff. If **EXP** = ABSTRACT[**v**,**BODY**], then it is not the case that SKI-NF-P[**EXP**] implies SKI-NF-P[**BODY**]. This is easy to see. Consider letting **BODY** be (I **v**), then **EXP** = S (K I) I. Therefore, SKI-NF-P[**EXP**], but (*not* SKI-NF-P[**BODY**]).

Theorem 1.12: *λ-TO-SKI preserves normal forms.* Let **EXP** ∈ λ-wff. If λ-NF-P[**EXP**] and **EXP′** = λ-TO-SKI[**EXP**], then SKI-NF-P[**EXP′**].

Proof:

The proof is by structural induction on **EXP**. There are three cases to consider:
Case 1: ATOM-P[**EXP**].
By definition of λ-TO-SKI, ATOM-P[**EXP′**].
ATOM-P[**EXP′**] implies SKI-NF-P[**EXP′**].
Case 2: **EXP** = (λ **bv BODY**). There are three sub-cases to consider.
Case 2a: **bv** = **BODY**.
By the definitions of λ-TO-SKI and ABSTRACT, **EXP′** = I.
ATOM-P[**EXP′**] implies SKI-NF-P[**EXP′**].
Case 2b: **bv** occurs in **BODY′** where **BODY′** = λ-TO-SKI[**BODY**].
COMBINATION-P[**BODY′**], for otherwise
BODY′ = **bv** = **BODY** which is case 2a.
By the definitions of λ-TO-SKI and ABSTRACT,
EXP′ = S **RATOR′ RAND′**,
where **RATOR′** = ABSTRACT[**bv**,OPERATOR[**BODY′**]] and
RAND′ = ABSTRACT[**bv**,OPERAND[**BODY′**]].
EXP′ = S **RATOR′ RAND′** implies
(*not* SKI-REDEX-P[**EXP′**]),
(*not* SKI-REDEX-P[OPERATOR[**EXP′**]]), and SKI-NF-P[S].
By definition of λ-NF-P, λ-NF-P[**BODY**].
By induction, SKI-NF-P[**BODY′**].
By definition of SKI-NF-P,
SKI-NF-P[OPERATOR[**BODY′**]] and SKI-NF-P[OPERAND[**BODY′**]].
By Theorem 1.11, then, SKI-NF-P[**RATOR′**] and SKI-NF-P[**RAND′**].
These facts imply SKI-NF-P[OPERATOR[**EXP′**]] and
SKI-NF-P[OPERAND[**EXP′**]].
Therefore, SKI-NF-P[**EXP′**].
Case 2c: **bv** does not occur in **BODY′** where **BODY′** = λ-TO-SKI[**BODY**].
By the definitions of λ-TO-SKI and ABSTRACT, **EXP′** = K **BODY′**, where
BODY′ = λ-TO-SKI[**BODY**].
(*not* SKI-REDEX-P[**EXP′**]).
By the definition of λ-NF-P, λ-NF-P[**BODY**].
By induction, SKI-NF-P[**BODY′**]. These facts imply SKI-NF-P[**EXP′**].
Case 3: **EXP** = **RATOR RAND**.
By the definitions of λ-TO-SKI and ABSTRACT, **EXP′** = **RATOR′ RAND′**, where
RATOR′ = λ-TO-SKI[**RATOR**] and
RAND′ = λ-TO-SKI[**RAND**].
By induction, SKI-NF-P[**RATOR′**] and SKI-NF-P[**RAND′**].
It remains to show that (*not* SKI-REDEX-P[**EXP′**]).
Assume SKI-REDEX-P[**EXP′**], then, by Lemma 1.4, β-REDEX-P[**EXP**].
β-REDEX-P[**EXP**] contradicts the hypothesis that λ-NF-P[**EXP**].
Hence (*not* SKI-REDEX-P[**EXP′**]).
Therefore SKI-NF-P[**EXP′**].

End Proof

Let **EXP** ∈ λ-wff. If **EXP′** = λ-TO-SKI[**EXP**], then it is not the case that SKI-NF-P[**EXP′**] implies λ-NF-P[**EXP**]. As an example, consider the λ-wff **EXP** = (λ y ((λ x x) y)). **EXP′** = S (K I) I. SKI-NF-P[**EXP′**] but (*not* λ-NF-P[**EXP**]).

Theorem 1.13: *Abstraction preserves lazy-normal form.* Let **v** ∈ VAR and **BODY** ∈ SKI-wff. If LAZY-NF-P[**BODY**] and **EXP** = ABSTRACT[**v**,**BODY**], then LAZY-NF-P[**EXP**].

Proof:

There are two cases to consider:
Case 1: ATOM-P[**BODY**]. There are two sub-cases to consider:
Case 1a: **v** = **BODY**.
By the definition of ABSTRACT, **EXP** = I.
EXP = I and ATOM-P[I] together imply LAZY-NF-P[**EXP**].
Case 1b: It is not the case that **v** = **BODY**.
By the definition of ABSTRACT, **EXP** = (K **BODY**).
EXP = K **BODY** implies (*not* SKI-REDEX-P[**EXP**]).
This and the fact that LAZY-NF-P[K] imply LAZY-NF-P[**EXP**].
Case 2: **BODY** = **RATOR RAND**. Again, there are two sub-cases to consider:
Case 2a: **v** occurs in **BODY**.
By the definition of ABSTRACT, **EXP** = S **RATOR′ RAND′**, where **RATOR′** = ABSTRACT[**v**,**RATOR**] and **RAND′** = ABSTRACT[**v**,**RAND**].
Since the SKI-wffs S, S **RATOR′**, and S **RATOR′ RAND′** are not SKI-redexes, LAZY-NF-P[S **RATOR′ RAND′**] — i.e. LAZY-NF-P[**EXP**].
Case 2b: **v** does not occur in **BODY**.
By the definition of ABSTRACT, **EXP** = (K **BODY**).
Same as case 1b.

End Proof

Let **EXP** ∈ λ-wff. If **EXP′** = λ-TO-SKI[**EXP**], then it is not the case that LAZY-NF-P[**EXP′**] implies **EXP** has a head-normal form. An example follows. Let **EXP** = (λ x ((λ y (y y)) (λ y (y y)))) which implies **EXP′** = K (S I I (S I I)). LAZY-NF-P[**EXP′**] but **EXP** has no head-normal form.

Let **EXP** ∈ λ-wff. If **EXP′** = λ-TO-SKI[**EXP**], then it is not the case that SKI-NF-P[**EXP′**] implies **EXP** has a λ-normal form. In fact, **EXP** may not even have a head-normal form. An example follows. Let **EXP** = (λ x ((λ z (z z x)) (λ z (z z x)))). The normal reduction sequence for **EXP** looks like:

λ x ((λ z (z z x)) (λ z (z z x))),
λ x ((λ z (z z x)) (λ z (z z x)) x)),
λ x ((λ z (z z x)) (λ z (z z x)) x x)),
λ x ((λ z (z z x)) (λ z (z z x)) x x x)),
...

EXP does not even have a head-normal form! But **EXP′** =

S (S (K (S (S I I))) (S (K K) I)) (S (K (S (S I I))) (S (K K) I))

does not contain any SKI-REDEXes! Therefore **EXP′** is in SKI-normal form.

Definition 1.65: Let **A** ∈ λ-wff. **A** is *in abs-normal form* iff *ABS-NF-P*[**A**] where
ABS-NF-P[**A**] $\overset{\text{Def}}{\rightarrow}$

```
(or VAR-P[A]
    ABSTRACTION-P[A]
    (and A = B C
        (not β-REDEX-P[A])
        ABS-NF-P[B]
        ABS-NF-P[C])).
```

Informally, a λ-wff is in abs-normal form if all of its occurrences of β-redexes lie in the bodies of abstractions.

Definition 1.66: Let **A** ∈ λ-wff. **A** is *in abs-head-normal form* iff (*ABS-HEAD-NF-P*[**A**]) where
ABS-HEAD-NF-P[**A**] $\overset{\text{Def}}{\rightarrow}$

```
(or VAR-P[A]
    ABSTRACTION-P[A]
    (and A = (B C)
        (not β-REDEX-P[A])
        ABS-HEAD-NF-P[B])).
```

Informally, a λ-wff is in abs-head-normal form if all of its occurrences of β-redexes occur either in the bodies of abstractions or in the operands of combinations which are not β-redexes themselves.

Theorem 1.14: Let **E** ∈ λ-wff. If **E**′ = λ-TO-SKI[**E**] and SKI-NF-P[**E**′], then ABS-NF-P[**E**].

Proof:

The proof is by structural induction on **E**′.

Case 1: ATOM-P[**E**′].

ATOM-P[**E**′] implies that either VAR-P[**E**′] or **E**′ = I.

If VAR-P[**E**′], then VAR-P[**E**] which implies ABS-NF-P[**E**].

In case **E**′ = I, **E** = (λ **v** **v**) for some variable **v**. Again, ABS-NF-P[**E**].

Case 2: **E**′ = **RATOR**′ **RAND**′.

E′ a combination implies that either **E** an abstraction or a combination

If **E** is an abstraction, then ABS-NF-P[**E**].

So, suppose **E** = **RATOR RAND**.

By definition of SKI-NF-P, both **RATOR**′ and **RAND**′ are in SKI-normal form.

By definition of λ-TO-SKI, **RATOR**′ = λ-TO-SKI[**RATOR**] and **RAND**′ = λ-TO-SKI[**RAND**].

By induction, both **RATOR** and **RAND** are in abs-normal form.

E is not a β-redex, for if it was, **E**′ would be an SKI-REDEX, by Lemma 1.4.

Therefore, ABS-NF-P[**E**].

End Proof

Let **EXP** ∈ λ-wff. If **EXP**′ = λ-TO-SKI[**EXP**], then it is not the case that ABS-NF-P[**EXP**] implies SKI-NF-P[**EXP**′]. Here's an example. Let **EXP** = (λ x ((λ y y) a)), which implies **EXP**′ = K (I a), which is not in SKI-NF.

Theorem 1.15: Let **EXP** ∈ λ-wff. If **EXP′** = λ-TO-SKI[**EXP**], then ABS-HEAD-NF-P[**EXP**] iff LAZY-NF-P[**EXP′**].

Proof:

First, suppose ABS-HEAD-NF-P[**EXP**]. To show: LAZY-NF-P[**EXP′**].
Shown by structural induction on **EXP**.
There are three cases to consider:
Case 1: ATOM-P[**EXP**].
By the definition of λ-TO-SKI, ATOM-P[**EXP′**].
ATOM-P[**EXP′**] implies LAZY-NF-P[**EXP′**].
Case 2: **EXP** = (λ **bv BODY**). There are three sub-cases to consider:
Case 2a: **bv** = **BODY**.
By the definitions of λ-TO-SKI and ABSTRACT, **EXP′** = I.
EXP′ = I and ATOM-P[I] together imply LAZY-NF-P[**EXP′**].
Case 2b: **bv** occurs in **BODY** (and COMBINATION-P[**BODY**]).
By the definitions of λ-TO-SKI and ABSTRACT, **EXP′** = S **RATOR′ RAND′**, where
BODY′ = λ-TO-SKI[**BODY**],
RATOR′ = ABSTRACT[**bv**,OPERATOR[**BODY′**]], and
RAND′ = ABSTRACT[**bv**,OPERAND[**BODY′**]].
(*not* SKI-REDEX-P[**EXP′**]),
(*not* SKI-REDEX-P[OPERATOR[**EXP′**]]), and LAZY-NF-P[S].
Therefore, by the definition of LAZY-NF-P, LAZY-NF-P[**EXP′**].
Case 2c: **bv** does not occur in **BODY**.
By the definitions of λ-TO-SKI and ABSTRACT, **EXP′** = K **BODY′**, where
BODY′ = λ-TO-SKI[**BODY**].
(*not* SKI-REDEX-P[**EXP′**]) and LAZY-NF-P[K].
These facts imply (by the definition of LAZY-NF-P) LAZY-NF-P[**EXP′**].
Case 3: **EXP** = **RATOR RAND**.
By the definition of λ-TO-SKI, **EXP′** = **RATOR′ RAND′**, where
RATOR′ = λ-TO-SKI[**RATOR**] and **RAND′** = λ-TO-SKI[**RAND**].
By the definition of ABS-HEAD-NF-P, ABS-HEAD-NF-P[**RATOR**].
By induction, LAZY-NF-P[**RATOR′**].
It remains to show (*not* SKI-REDEX-P[**EXP′**]).
Assume SKI-REDEX-P[**EXP′**], then, by Lemma 1.4, β-REDEX-P[**EXP**].
β-REDEX-P[**EXP**] contradicts the hvpothesis that ABS-HEAD-NF-P[**EXP**].
Hence (*not* SKI-REDEX-P[**EXP′**]).
Therefore, LAZY-NF-P[**EXP′**].
It has been shown ABS-HEAD-NF-P[**EXP**] implies LAZY-NF-P[**EXP′**].

Now suppose LAZY-NF-P[**EXP′**]. To show: ABS-HEAD-NF-P[**EXP**].
Shown by structural induction on **EXP′**.
Case 1: ATOM-P[**EXP′**].
ATOM-P[**EXP′**] implies that either VAR-P[**EXP′**] or **EXP′** = I.
If VAR-P[**EXP′**], then VAR-P[**EXP**] which implies ABS-HEAD-NF-P[**EXP**].
In case **EXP′** = I, **EXP** = (λ **v v**) for some variable **v**.
Again, it is the case that ABS-HEAD-NF-P[**EXP**].
Case 2: **EXP′** = **RATOR′ RAND′**.
EXP′ a combination implies that either **EXP** an abstraction or a combination.
If **EXP** is an abstraction, then ABS-HEAD-NF-P[**EXP**].
So suppose **EXP** = **RATOR RAND**.

By definition of LAZY-NF-P, **RATOR′** in lazy-normal form.
By definition of λ-TO-SKI, **RATOR′** = λ-TO-SKI[**RATOR**].
By induction, **RATOR** is in abs-normal form.
EXP is not a β-redex, for if it was, **EXP′** would be an SKI-REDEX, by Lemma 1.4.
Therefore, ABS-HEAD-NF-P[**EXP**].
It has been shown that LAZY-NF-P[**EXP′**] implies ABS-HEAD-NF-P[**EXP**].
Therefore ABS-HEAD-NF-P[**EXP**] iff LAZY-NF-P-[**EXP′**].
End Proof

It is not the case for an arbitrary SKI-wff **E′** in SKI-NF that SKI-TO-λ[**E′**] is in ABS-NF. For example, let **E′** = (K z). **E′** is in SKI-NF. It is not the case, however, that SKI-TO-λ[**E′**] = ((λ x (λ y x)) z) in ABS-NF. This same example demonstrates that SKI-TO-λ does NOT "preserve redexes".

Conjecture 1.1: Let **A** ∈ SKI-wff. If **A′** = SKI-TO-λ[**A**] and SKI-NF-P[**A**], then **A′** has an abs-normal form.

The following result is an immediate consequence of the previous theorem. It is included here for completeness.

Theorem 1.16: *λ-TO-SKI preserves quasi-normal forms.* Let **EXP** ∈ λ-wff. If HEAD-NF-P[**EXP**] and **EXP′** = λ-TO-SKI[**EXP**], then LAZY-NF-P[**EXP′**].
Proof:
From the definitions of HEAD-NF-P and ABS-HEAD-NF-P,
it is clear that HEAD-NF-P[**EXP**] implies ABS-HEAD-NF-P[**EXP**].
By Theorem 1.15, then, LAZY-NF-P[**EXP′**].
End Proof

The relationship between SKI-wffs in lazy-normal form and λ-wffs has been demonstrated formally. The counterpart wffs in the λ-calculus to SKI-wffs in lazy-normal form are the λ-wffs in abs-head-normal form. In a later chapter it will be argued that, when reducing, "stopping at" lazy-normal form, rather than continuing on to SKI-normal form, has many computational advantages.

1.5. The λ-G-calculus

The λ-G-calculus, presented in [Wadsworth 1971], is a deterministic graph oriented version of Church's λ-calculus. That is, well-formed formulas in the λ-G-calculus are rooted acyclic graphs as opposed to strings in the λ-calculus.

The Standardization Theorem for the λ-calculus guarantees that if a λ-wff has a λ-normal form then it can be reached by a λ-normal reduction sequence. Unfortunately, performing λ-normal reductions on strings often causes duplication of redexes, thus creating more work than necessary. Using graphs as well-formed formulas instead of strings, Wadsworth was able to reduce (but not eliminate) the number of duplicated redexes that arise when performing λ-normal reductions.

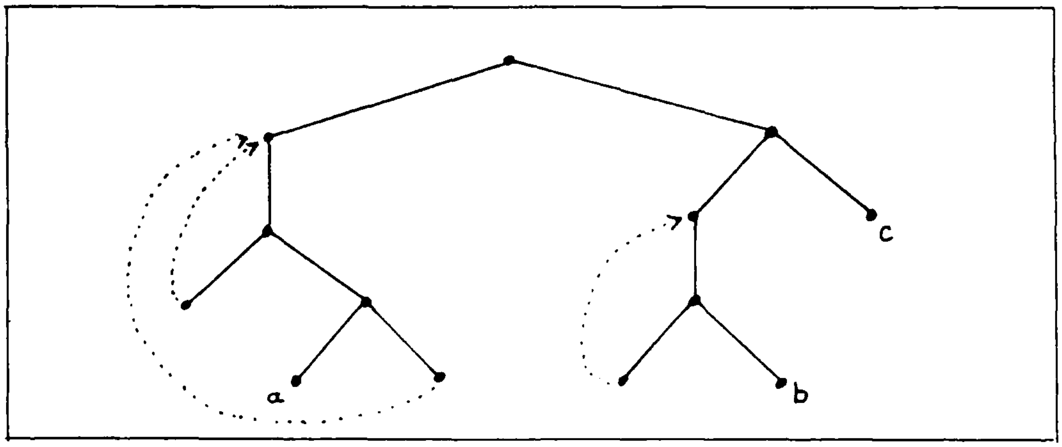
What follows is an informal account of Wadsworth's λ-G-calculus and his suggested implementation of it. For a formal description of the calculus, the reader is encouraged to read Chapter 4 of Wadsworth's thesis, [Wadsworth 1971].

1.5.1. Well-formed Formulas

Free variable occurrences are terminal nodes in the graph labeled with the name of the variable.

A combination is a graph whose root node has two outgoing arcs. One arc points at the graph which is the combination's operator and the other points at the graph which is the combination's operand.

An abstraction is a graph whose root node has a single outgoing arc. The arc points at the graph which is the body of the abstraction. Free occurrences of the abstraction's bound variable in the body are nodes which point back to the root node of the abstraction. These "back pointing" arcs, emanating from the bound variable nodes, are treated specially (see next section). Think of them as dotted arcs (lines) and the other arcs as solid. It was stated in the introduction to this calculus that these graphs were acyclic. That statement was a simplification of the truth. The truth is that the only cycles in the graph are those containing exactly one dotted arc.

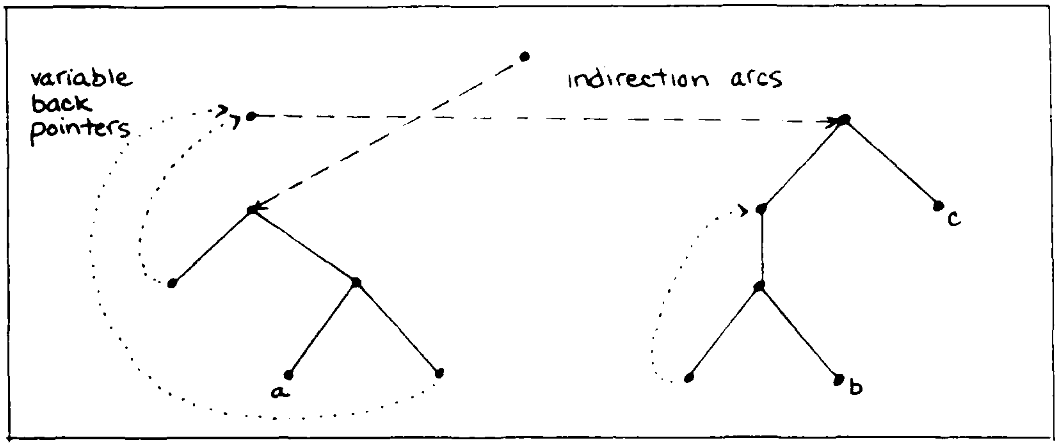


The λ-G-wff equivalent of the λ-wff: (λ x (x (a x))) ((λ z (z b)) c)
Figure 1.1

Some liberties were taken in the preceding description of Wadsworth's wffs. In Wadsworth's thesis the back pointers were not part of the formal calculus — they were introduced as an efficient representation for bound variable nodes in his implementation of the calculus. In his formal description, bound variable nodes looked just like free variable nodes. One determined that they were bound by seeing if there was a path from an abstraction node (labeled with the name of the variable it was binding) to it and making sure that the variable names were the same.

1.5.2. Reduction

β-reduction is performed in the λ-G-calculus by pointer manipulation rather than by string substitution.



The λ-G-wff in Figure 1.1 after contracting leftmost redex
Figure 1.2

Note that the redex ((λ z (x b)) c) is not duplicated (as would have happened if the equivalent reduction of the λ-wff had been performed). Instead, the redex is now being shared by two portions of the reduced λ-G-wff.

To accomplish this reduction, the two following operations were performed:

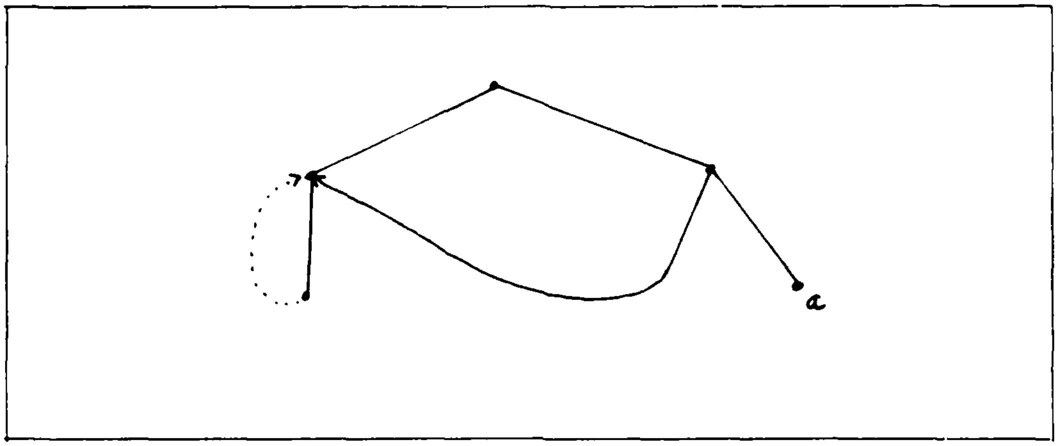
1. An indirection arc (different from both the solid arcs and the dotted arcs described above) was drawn from the root of the wff to the body of the abstraction. This new kind of arc is represented by a dashed line in the figure.
2. Another indirection arc was drawn from the root of the abstraction to the operand

Observe that is not necessary to search the body of the β-redex's operator (abstraction) for the free occurrences of the abstraction's bound variable to perform the contraction.[2]

When the algorithm "sees" a node (n_1) which has been "forwarded" via an indirection arc to another node (n_2), it ignores node n_1 and, instead, "sees" node n_2 — the node n_1 was forwarded to. Variable nodes which have (dotted) arcs emanating from them (the bound variables) are similarly ignored if the abstraction node to which they point has been forwarded. Variable nodes which point back to abstraction nodes which have not been forwarded are treated as terminal nodes in the graph.

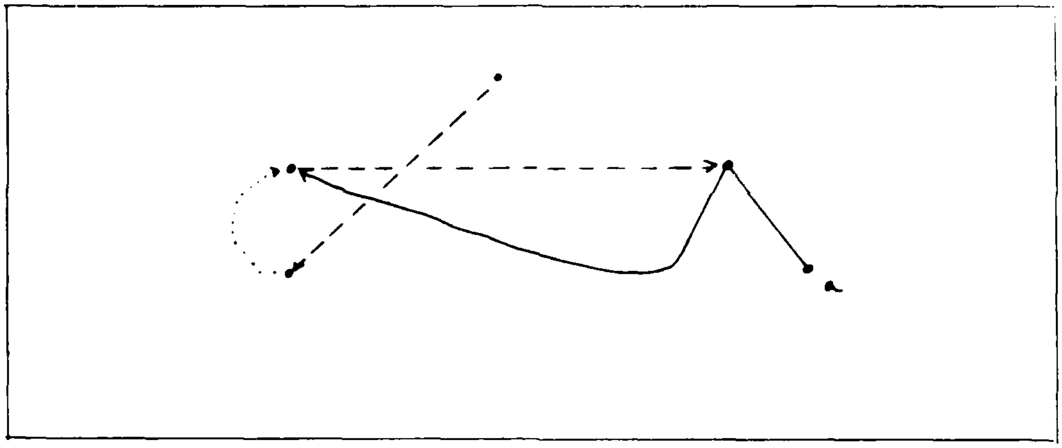
This simple version of λ-normal β-reduction of λ-G-wffs will not suffice in all situations. In the case where the operator (the abstraction) of the β-redex is pointed at by more than one node (not counting the bound variable back pointers), a portion of the abstraction's body must be copied before the contraction can take place. If this copying is not performed, erroneous results may occur. As an example of this situation, observe the following λ-G-wff:

[2] Arvind, in a paper which reviews several graph oriented interpreters ([Arvind 1984]), incorrectly states that all leaves of the operator must be searched for occurrences of the abstraction's bound variable. Arvind (mistakenly) thinks that many (one for each bound variable) indirection arcs to the operand are placed in the body of the operator. Instead, just one indirection arc, from the abstraction's root to the operand, is required.



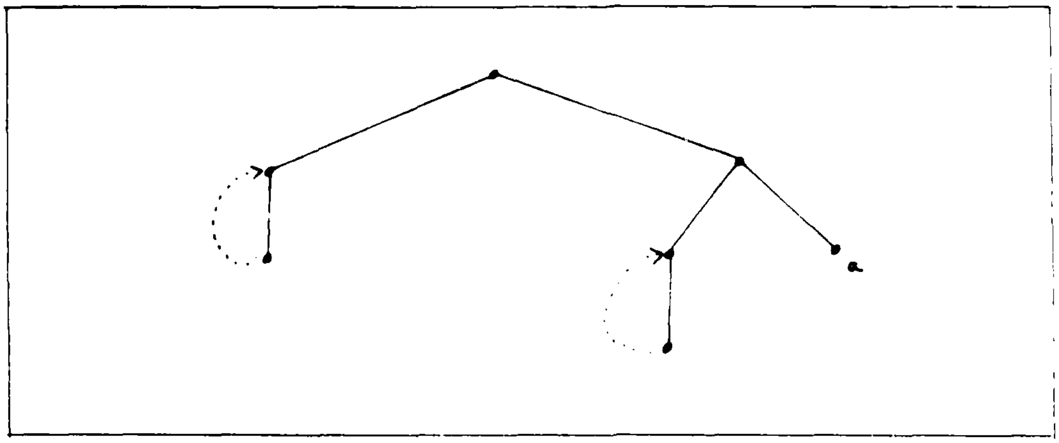
A β-reduction cannot be safely performed on this λ-G-wff
Figure 1.3

If a β-reduction of the type described above were performed on the λ-G-wff in Figure 1 3, then the result would not be a λ-G-wff at all! The result would be the following graph:



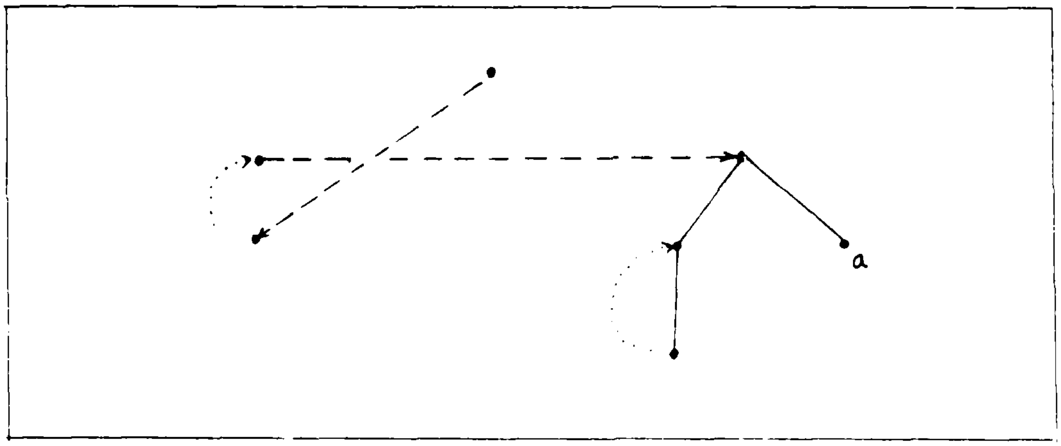
Note the cycles in this non λ-G-wff
Figure 1.4

In order to insure a proper β-contraction, some copying must take place before the contraction is attempted



The λ-G-wff in Figure 1.3 after copying
Figure 1.5

A β-reduction may be safely performed on the graph displayed above The result is the λ-G-wff:



After performing β-reduction
Figure 1.6

The parts of the body which do not contain free occurrences of the bound variable are called the abstraction's *free expressions* Free expressions which are not contained in any of the abstraction's other free expressions are called the abstraction's *maximal free expressions*; this name was given to them later in [Hughes 1982a]. These maximal free expressions of the operator need not be copied before performing the contraction. Wadsworth's interpreter is called *fully lazy* since it performs normal order graph reduction (making it lazy) and avoids repeated reduction of constant expressions (since they are not copied).

Observe that since some copying must be done, when a redex exists in the expressions copied, it will be copied. Wadsworth's calculus, therefore, is not optimal -- i.e. there may be shorter reduction sequences ending in normal form. For example, consider the expression:

$$(\lambda\ x\ (x\ x))(\lambda\ y\ ((\lambda\ \mathbf{z}\ \mathbf{z})\ y)).$$

which, when reduced to normal form in Wadsworth's calculus, takes four steps (because the boldface redex must be copied). If, however, the boldface redex is reduced first, then it can be reduced to normal form in only three steps.[3]

1.6. Summary

Three reduction calculi have been described: the λ-calculus, the SKI-calculus, and the λ-G-calculus. The λ-calculus looks the most like a programming language. The SKI-calculus is the simplest. The λ-G-calculus appears to be the most implementation oriented.

In the next chapter, two more calculi are presented: the SKI-G-calculus and the LNF-calculus. Both are deterministic and "machine oriented". The SKI-G-calculus is a graph oriented version of the SKI-calculus. The LNF-calculus is also graph oriented but contains many more functors and a new class of atomic wffs called constructors. This richer calculus, when realized, yields an efficient runtime system for the LNF language. The runtime system's implementation is detailed in Chapter 3, Section 4.

[3] Wadsworth, in his thesis, also points out that his calculus is nonoptimal. Unfortunately the example he presents ([Wadsworth 1971], page 187) which purports to demonstrate this fact does not do so.

Chapter 2

Two Deterministic Graph Oriented Reduction Calculi

The LNF Language's run-time system (its Lisp Machine implementation is detailed in Chapter 3) is a realization of a deterministic reduction calculus called the LNF-calculus. The LNF-calculus is based on another deterministic reduction calculus called the SKI-G-calculus. Both calculi are given formal definitions in this chapter.

The SKI-G-calculus is presented first. The SKI-G-calculus, like Wadsworth's λ-G-calculus ([Wadsworth 1971]), is graph oriented. Instead of being based on the λ-calculus, however, the SKI-G-calculus is a modification of the SKI-calculus.

In essence, the SKI-G-calculus is a formalization of the "normal order combinator graph reduction" machine informally described in [Turner 1979c]. The calculus' description. although similar in style to Wadsworth's description of the λ-G-calculus, is much more "machine oriented" than Wadsworth's. For example, Wadsworth relegates forwarding arcs — forwarding arcs are also often referred to as indirection pointers or invisible pointers — to his implementation of the calculus and does not even mention garbage nodes in his discussions. On the other hand, in the SKI-G-calculus, garbage vertices and forwarding arcs are given formal definitions. The definitions of SKI-G-wff and SKI-G-mr, taken together, come very close to being an implementation of the SKI-G-calculus as well as its definition.

It is claimed, but not proved, that the (deterministic) SKI-G-calculus is computationally equivalent to the (nondeterministic) SKI-calculus (and, of course, to the λ-calculus et al.)

As stated above, the LNF-calculus is based on the SKI-G-calculus. Its set of wffs (LNF-wff) contains SKI-G-wff. LNF-wff contains SKI-G-wff by virtue of the fact that LNF-calculus' set of functors (combinators, primitive operators) contains SKI-G-calculus' functor set. The LNF-calculus has, in addition to Schönfinkel's functors S, K, and I ([Schönfinkel 1924]); Curry's B, C, and W ([Curry 1958]); Turner's S′ and C′, Scheevel's B′ ([Turner 1979a] and [Turner 1984]); numeric functors, boolean functors, and a few others of the author's design. Besides the addition of these new functors, new atoms, called constructors, are introduced into LNF-wff.

The "immediately reducible to" relation of the LNF-calculus (LNF-imr) differs from SKI-G-imr in the following three ways. Firstly, LNF-imr does not contain SKI-G-imr — i.e. there are wffs which are reducible in the SKI-G-calculus but irreducible in the LNF-calculus. These are exactly those wffs in SKI-G-lazy-normal form (containing no initial redex) but containing redexes elsewhere. In sum, many of the reduction contexts present in the SKI-G-calculus do not exist in the LNF-calculus. Recall that a reduction context is a context inside which a reduction is permitted to take place. These reduction contexts are specified by the contextual reduction rules of a calculus. Secondly, the new functors bring with them new ways of reducing the LNF-wffs having them as initial atoms — via new substantive reduction rules. Lastly, the new functors ("making up for" the lack of general reduction contexts present) bring with them new "functor specific" reduction contexts — via new contextual reduction rules. The end result is a lazy "immediately reducible to" relation which allows "just enough reduction to get the job done". The addition of the constructors does not substantively affect the "immediately reducible to" relation. However, their addition (by increasing the size of the set of well-formed formulas) indirectly extends LNF-imr.

The LNF-calculus, of course, does not have any more computational power than the other calculi defined herein — it is, however, a few steps nearer the "directly and efficiently implementable" end of the reduction calculus spectrum than the others. It is hoped that a calculus which bridges the gap between traditionally defined formal calculi and their implementations will be easier to implement and its implementation easier to reason about.

The notions of initial-redex and lazy-normal form, as defined in the SKI-calculus, have corresponding definitions in the SKI-G-calculus and the LNF-calculus. These concepts figure prominently in the organization of the two calculi.

2.1. The SKI-G-calculus

The SKI-G-calculus is a graph oriented version of Schönfinkel's SKI-calculus.

2.1.1. Well-formed Formulas

As SKI-G-calculus well-formed formulas (SKI-G-wffs) are defined in terms of graphs, the graph related conventions which will be used are described below.

A graph is defined by a set of vertices and a set of arcs. Identifiers denoting vertices are written in lowercase while identifiers representing sets of vertices are written in uppercase. Just as in the preceding chapter, wffs are also denoted by uppercase identifiers. An arc having origin $\mathbf{v}_1$ and destination $\mathbf{v}_2$ is written as the ordered pair $<\mathbf{v}_1,\mathbf{v}_2>$. Paths are sequences of arcs (possibly empty) of the form:

$$<\mathbf{v}_1,\mathbf{v}_2>,<\mathbf{v}_2,\mathbf{v}_3>, \ldots, <\mathbf{v}_{n-2},\mathbf{v}_{n-1}>,<\mathbf{v}_{n-1},\mathbf{v}_n>.$$

A vertex $\mathbf{v}_n$ is said to be accessible from $\mathbf{v}_1$ if there is a path from $\mathbf{v}_1$ to $\mathbf{v}_n$. Hence, each vertex is accessible from itself via the path of length 0. For rooted graphs **G** (those which contain a vertex designated as the root), the set of vertices accessible from the root is represented by the expression *ACCESSIBLE-VS*[G].

Definition 2.1: An *SKI-G-wff* **X** is a finite rooted graph represented by the sextuple <**VS**,**RATOR**,**RAND**,**FWD**,**ATOM**,**root**> where:

```
(and
  VS is the (finite) set of vertices of X
  RATOR, RAND, and FWD are sets of arcs —
    together these sets partition the set of arcs of X
  ATOM is a nonempty partial function from VS to {S,K,I}
  root is the vertex in VS designated as X's root
  For all vertices v ∈ VS,
   (and the out degree of v is either 0, 1, or 2
        in case v's out degree is 0
          then ATOM[v] defined
        in case v's out degree is 1, then
          (and the arc having origin v lies in FWD
               ATOM[v] undefined)
        in case v's out degree is 2, then
          one of the arcs having origin v lies
            in RATOR, the other in RAND, and
            ATOM[v] undefined,
        there is no non-empty path from v to v,
          all the arcs of which are in FWD)
  there is a v ∈ VS such that:
   (and v is accessible from root
        v has out degree 0 or 2))
```

Note that variables are not a part of this calculus. They have been excluded as only closed λ-wffs are transformed into SKI-G-wffs. Well-formed LNF programs will not contain occurrences of free variables. Since the transformation replaces all occurrences of bound variables with SKI-G-wffs not containing variables, and there are no free occurrences of variables in the λ-wff being transformed (it is closed), the resulting SKI-G-wff will not contain any variables at all.

Definition 2.2: Let **X** = <**VS**,**RATOR**,**RAND**,**FWD**,**ATOM**,**root**> be an SKI-G-wff.

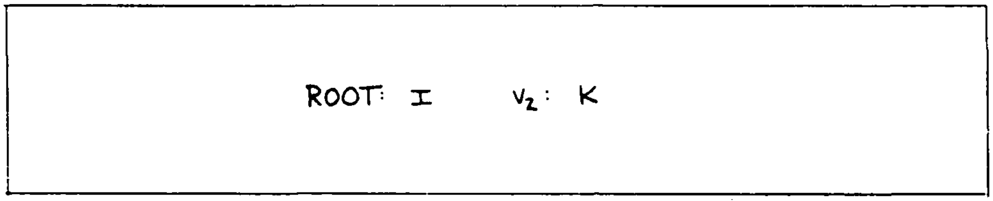
- The *root of* **X** (*ROOT*[**X**]) is **root**.
- The *set of vertices of* **X** (*VS*[**X**]) is VS.
- The *rator arc set of* **X** (*RATOR*[**X**]) is RATOR.
- The *rand arc set of* **X** (*RAND*[**X**]) is RAND
- The *forwarding arc set of* **X** (*FWD*[**X**]) is FWD
- The *atom function of* **X** (*ATOM*[**X**]) is ATOM.

Note that the definition of SKI-G-wff does not require that each vertex of an SKI-G-wff be accessible from the SKI-G-wff's root It does require, however, that all vertices in an SKI-G-wff's vertex set, accessible or not, be eligible for "roothood" — i.e. let **X** be an SKI-G-wff and let **v** be any vertex in VS[**X**]. It can be shown that the graph, which is just like the SKI-G-wff **X** except that it has v for a root, also qualifies as an SKI-G-wff.

Definition 2.3: Let **X** be an SKI-G-wff. The vertices in VS[**X**] which are inaccessible from **X**'s root (not in ACCESSIBLE-VS[**X**]) are the *garbage of* **X**. This set of vertices is denoted by the expression *GARBAGE*[**X**].

Definition 2.4: Let **X** be an SKI-G-wff. **X** is *clean* (*CLEAN-P*[**X**]) iff VS[**X**] = ACCESSIBLE-VS[**X**].

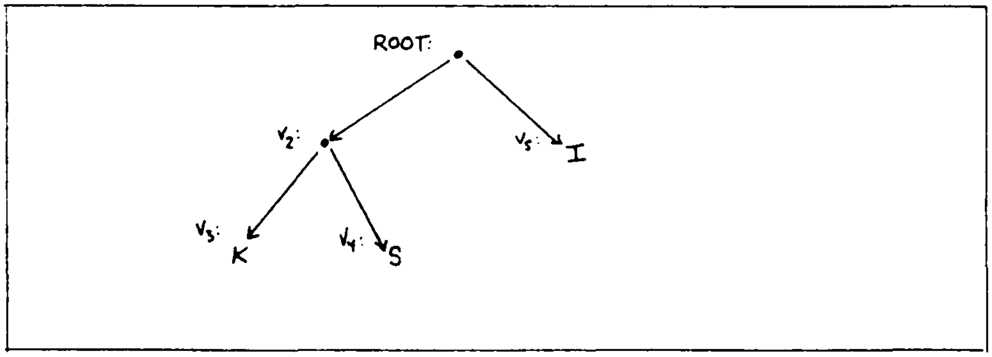
An SKI-G-wff <**VS**,**RATOR**,**RAND**,**FWD**,**ATOM**,**root**> is represented on paper as follows. A vertex **v** having out degree 0 is represented by the functor **ATOM**[**v**]. A vertex **v** having out degree 1 (a forwarding vertex) is represented by a dot (•) having one dotted arrow (representing the arc <v,**fwdv**> in **FWD**) pointing at the representation of **fwdv**. A vertex having out degree 2 is represented by a dot having two arrows - representing the two arcs which emanate from it <**v**,**rtr**> (left arrow) and <**v**,**rnd**> (right arrow) - which point at the representations of **rtr** and **rnd**. The vertex **root** is often labeled with the string "ROOT:". Often other vertices are given labels to ease reference. See the figures below for some examples of this representation.



The SKI-G-wff: $<\{v_1,v_2\},\{\},\{\},\{\},\{<v_1,I>,<v_2,K>\},v_1>$

Figure 2.1

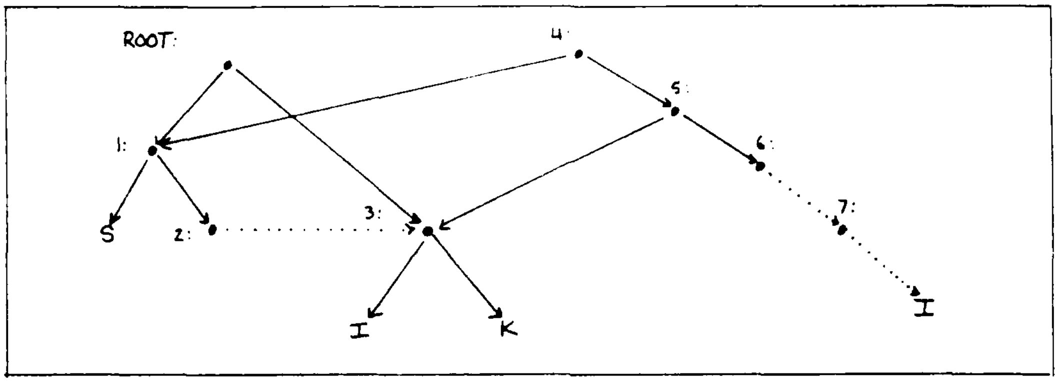
Note that the vertex v_2 in the above diagram is a garbage vertex. It is garbage since it is inaccessible from the root (v_1).



The Clean SKI-G-wff:

$<\{v_1,v_2,v_3,v_4,v_5\},\{<v_1,v_2>,<v_2,v_3>\},\{<v_1,v_5>,<v_2,v_4>\},\{\},\{<v_3,K>,<v_4,S>,<v_5,I>\},v_1>$

Figure 2.2



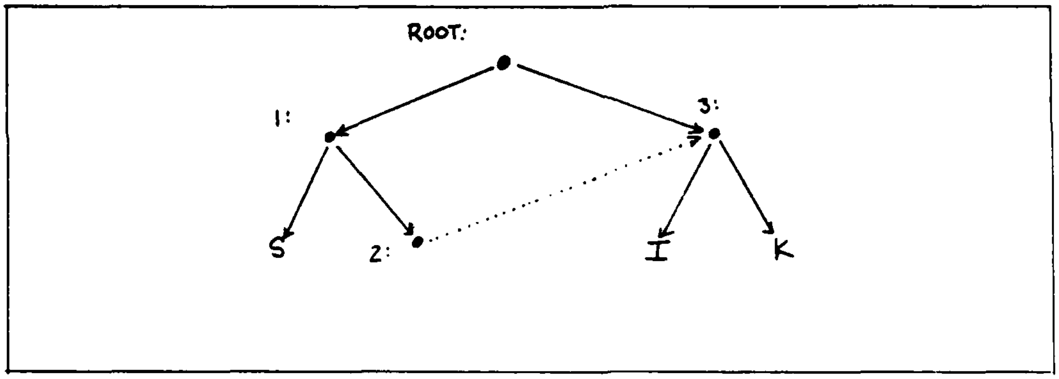
An SKI-G-wff with Some Forwarding Vertices and Some Shared Subformulas
Figure 2.3

This representation is a good one as it allows one to observe the SKI-G-wff's structure at a glance. With it one can easily identify a wff's garbage, root, shared subformulas, and cycles. Often, however, because this representation is so difficult to typeset, SKI-G-wffs are displayed linearly — just like SKI-wffs. When using this linear representation, garbage and forwarding nodes are ignored completely, shared structures are undetectable, and cycles are unrepresentable. This linear display is used only when these aspects of the wff are not important.

Let $\langle$**VS**,**RATOR**,**RAND**,**FWD**,**ATOM**,**root**$\rangle$ be an SKI-G-wff. Viewing the arc sets **RATOR**, **RAND**, and **FWD** as functions from vertices to vertices is sometimes *useful. Let* **S** *be either* **RATOR**, **RAND**, *or* **FWD**. *For all arcs* $\langle \mathbf{v}_1,\mathbf{v}_2 \rangle$ *in* **S**, $\mathbf{S}[\mathbf{v}_I]$ $= \mathbf{v}_2$. Let **F** be a function with domain **D**. If **SD** is a subset of **D**, then the restriction of **F** to sub-domain **SD** is written **F**|**SD**.

Definition 2.5: Let **X** be an SKI-G-wff. *The clean SKI-G-wff in* **X** is *CLEAN*[**X**] where:
CLEAN[**X**] $\overset{\text{Def}}{\Rightarrow}$

$\langle$**VS′**,**RATOR′**,**RAND′**,**FWD′**,**ATOM′**,ROOT[**X**]$\rangle$
where
VS′ *is* ACCESSIBLE-VS[**X**] &
RATOR′ *is* RATOR[**X**]|**VS′** &
RAND′ *is* RAND[**X**]|**VS′** &
FWD′ *is* FWD[**X**]|**VS′** &
ATOM′ *is* ATOM[**X**]|**VS′**



CLEANed Version of the SKI-G-wff in Figure 2.3
Figure 2.4

The definition of the function CLEAN might be viewed as a very high level specification of a garbage collector. By providing different realizations of the predicate ACCESSIBLE and the function restricting operator |, one is able to create different implementations of the specification.

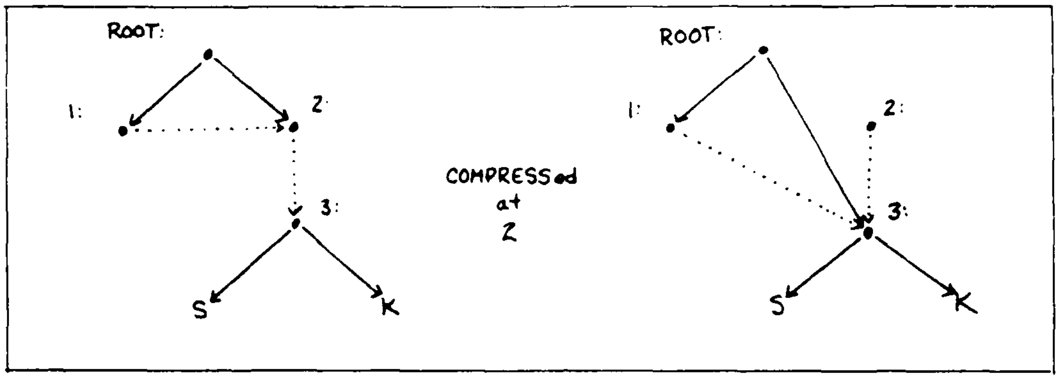
Definition 2.6: Let **X** be an SKI-G-wff and let **v** be a vertex in VS[**X**]. The vertex **v** *is forwarded to* **v'** *in* **X** (also *FORWARDED-P*[v,X], *FORWARDED-TO*[v,X] = v') iff <v,v'> ∈ FWD[X].

Definition 2.7: Let **X** be an SKI-G-wff. **X** is *compact* (*COMPACT-P*[X]) iff for all vertices v ∈ VS[X], FORWARDED-P[v,X] implies v ∈ GARBAGE[X].

The following definition defines a function (COMPRESS) which removes one source of indirection in an SKI-G-wff containing a forwarding arc. It does so by replacing all arcs which point at the forwarded vertex with arcs which point at the vertex to which the forwarded vertex points.

Definition 2.8: Let **X** be an SKI-G-wff. Let v ∈ VS[X] such that FORWARDED-P[v,X]. *The SKI-G-wff contained in* **X** *compressed at* v is *COMPRESS*[v,X], where COMPRESS[v,X] $\overset{\text{Def}}{\rightarrow}$

```
<VS[X],RATOR,RAND,FWD,ATOM[X],root>
 where
   RATOR is RATOR[X]
     with all arcs of the form <u,v>
     replaced with <u,FORWARDED-TO[v,X]> &
   RAND is RAND[X]
     with all arcs of the form <u,v>
     replaced with <u,FORWARDED-TO[v,X]> &
   FWD is FWD[X]
     with all arcs of the form <u,v>
     replaced with <u,FORWARDED-TO[v,X]> &
   root is (if (not ROOT[X] = v)
              then ROOT[X]
             else FORWARDED-TO[v,X])
```



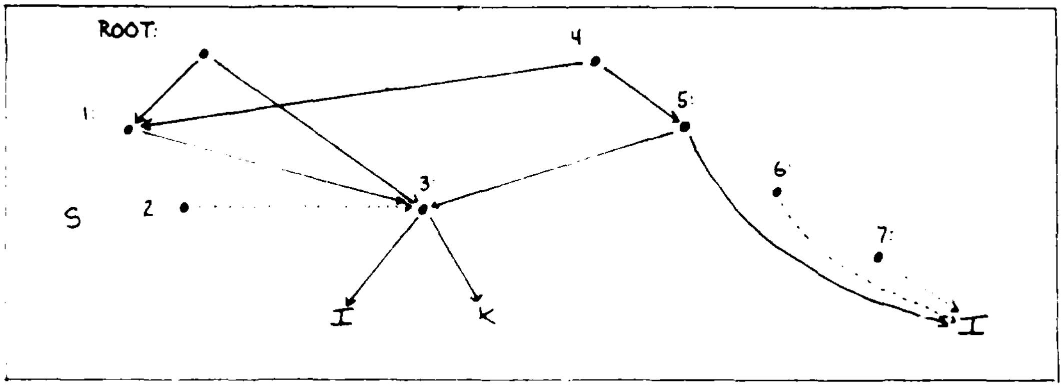
An Example of COMPRESSion
Figure 2.5

Note that although all of the arcs whose destination had been the forwarding vertex have been removed, the forwarding vertex and its forwarding arc have not. The forwarding vertex is now inaccessible from the root of the new SKI-G-wff. It therefore is part of the garbage of the compressed SKI-G-wff.

The next function (COMPACT), defined in terms of COMPRESS, makes all sources of indirection (all forwarding vertices) into garbage.

Definition 2.9: Let **X** be an SKI-G-wff. *The compact SKI-G-wff contained in* **X** is (*COMPACT*[**X**]) where:

```
COMPACT[X] $\overset{Def}{\rightarrow}$
  (if COMPACT-P[X]
    then X
   else
    (let v be
      (a vertex in VS[X] such that there is
        an arc <v,vfwd> in FWD[X])
      in COMPACT[COMPRESS[v,X]]))
```



COMPACTed Version of the SKI-G-wff in Figure 2.3
Figure 2.6

Although not proved here, it can be shown that the functions CLEAN and COMPACT really do produce SKI-G-wffs. As will be seen in subsequent chapters, the LNF compiler produces clean compact SKI-G-wffs (actually, it produces clean compact LNF-wffs) from user input.

Definition 2.10: Let **X** be an SKI-G-wff. **X** is a *combination* (*COMBINATION-P*[X]) iff there is an arc in RATOR[X] (which implies there is an arc in RAND[X] also) whose origin is in ACCESSIBLE-VS[X]. **X** is an *atom* (*ATOM-P*[X]) iff it is not a combination.

Note that an atomic SKI-G-wff (a wff **A** such that ATOM-P[A]) may contain more than one accessible vertex. There may be a path, composed exclusively of forwarding arcs, from the root to a vertex which is mapped by the wff's atom function to one of the functors: S, K, or I.

Definition 2.11: Let **X** be an SKI-G-wff and let **v** be in VS[X]. The *SKI-G-wff described in* **X** *rooted at* **v** is (*SKI-G-WFF*[X,v]) where
SKI-G-WFF[X,v] $\stackrel{\text{Def}}{\rightarrow}$
<VS[X],RATOR[X],RAND[X],FWD[X],ATOM[X],v>

If **v** in (ACCESSIBLE-VS[X]), then SKI-G-WFF[X,v] is called *the subformula of* **X** *rooted at* **v** or SUBFORMULA[X,v].

The subformula of an SKI-G-wff **X** rooted at v (call it **X'**) is often referred to as, simply, a subformula of **X**. It is also said that **X** *contains* **X'** or **X'** *occurs in* **X**. It is important to observe that for any SKI-wff **X** and any **Y** which is a subformula of **X** the sets VS[X] and VS[Y] are identical. Besides the subformulas of **X**, there are other SKI-G-wffs described by **X**. These are the SKI-G-wffs which are rooted at the vertices in GARBAGE[X].

Definition 2.12: Let **X** be an SKI-G-wff. If **X** is a combination, then there are two (not necessarily distinct) *immediate subformulas of* **X**:

```
OPERATOR[X] $\stackrel{\text{Def}}{\rightarrow}$
  (if FORWARDED-P[ROOT[X],X]
    then OPERATOR[SUBFORMULA[X,FORWARDED-TO[ROOT[X],X]]]
   else SUBFORMULA[X,RATOR[X][ROOT[X]]])

OPERAND[X] $\stackrel{\text{Def}}{\rightarrow}$
  (if FORWARDED-P[ROOT[X],X]
    then OPERAND[SUBFORMULA[X,FORWARDED-TO[ROOT[X],X]]]
   else SUBFORMULA[X,RAND[X][ROOT[X]]])
```

Observe that RATOR[X][ROOT[X]] (RAND[X][ROOT[X]]) is the result of applying the function RATOR[X] (RAND[X]) to the vertex specified by ROOT[X].

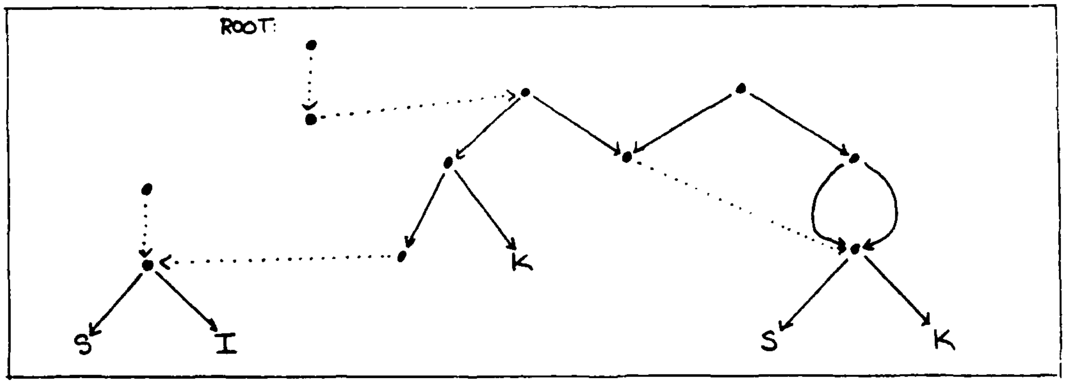
It is hoped that no confusion will arise due to the author's overloading of the predicates: COMBINATION-P and ATOM-P, and the functions: OPERATOR and OPERAND. It should always be clear from the context which calculus, and therefore which predicate (or function), is being referenced.

Definition 2.13: Let **X** be a combination. Let **X**$'$ be the subformula of **X** rooted at v. If there is more than one path from ROOT[X] to v in **X**, then **X**$'$ is *a shared subformula of* **X** (*SHARED-P*[X$'$,X]).

Definition 2.14: An SKI-G-wff **X** *contains a cycle* if there is a path (having length greater than 0) from an accessible vertex v to itself.

Definition 2.15: If an SKI-G-wff **X** does not contain any cycles, then applying the function *GRAPH-TO-STRING* to **X** yields an SKI-wff (called the *linear transform of* **X**).

```
GRAPH-TO-STRING[X] $\overset{\text{Def}}{\rightarrow}$
  (let root be ROOT[X] in
   (if FORWARDED-P[root,X]
     then GRAPH-TO-STRING[SUBFORMULA[X,FORWARDED-TO[root,X]]]
    elseif ATOM-P[X]
     then ATOM[X][root]
    else ;; X is a combination
     (GRAPH-TO-STRING[OPERATOR[X]] GRAPH-TO-STRING[OPERAND[X]])))[1]
```



S I K (S K) is the Linear Transform of the Above SKI-G-wff
Figure 2.7

Note that an SKI-G-wff's garbage is not a factor in this transformation. Also, forwarding vertices and their arcs are used only as "indirection pointers" by GRAPH-TO-STRING. Any shared subformula in the SKI-G-wff is transformed into multiple occurrences of the subformula in the SKI-wff.

Definition 2.16: Let **X** and **Y** be acyclic SKI-G-wffs. **X** *is synonymous with* **Y** iff (*SYNONYMOUS-P*[X,Y]) where

SYNONYMOUS-P[X,Y] $\overset{\text{Def}}{\rightarrow}$

GRAPH-TO-STRING[X] = GRAPH-TO-STRING[Y]

[1] On the confusing syntax — the two preceding right parentheses are part of the syntax of the definition, while the left and right parentheses enclosing the expressions GRAPH-TO-STRING[OPERATOR[X]] and GRAPH-TO-STRING[OPERAND[X]] are part of the result.

Theorem 2.1: Any acyclic SKI-G-wff may be COMPACTed and then CLEANed to produce a clean compact synonymous SKI-G-wff. The proof follows directly from the definitions of the functions CLEAN, COMPACT, and GRAPH-TO-STRING.

Definition 2.17: Let **a** be a functor (which is an SKI-wff). The *atomic graphical transform of* **a** is *ATOMIC-GRAPH*[**a**] where:
ATOMIC-GRAPH[**a**] $\overset{\text{Def}}{\rightarrow}$

```
(let nv be a new vertex in
  <{nv},{},{},{},{<nv,a>},nv>)
```

Definition 2.18: Let **X** and **Y** be SKI-G-wffs. **X** and **Y** are *compatible* if *COMPATIBLE-P*[**X**,**Y**] where:
COMPATIBLE-P[**X**,**Y**] $\overset{\text{Def}}{\rightarrow}$

```
(or VS[X] ∩ VS[Y] = ∅
    X is a subformula of Y
    Y is a subformula of X)
```

Definition 2.19: Let **X** and **Y** be compatible SKI-G-wffs. The *combination of* **X** and **Y** is *COMBINE*[**X**,**Y**] where:
COMBINE[**X**,**Y**] $\overset{\text{Def}}{\rightarrow}$

```
(let root be a new vertex in
  <VS[X] ∪ VS[Y] ∪ {root},
    RATOR[X] ∪ RATOR[Y] ∪ {<root,ROOT[X]>},
    RAND[X] ∪ RAND[Y] ∪ {<root,ROOT[Y]>},
    FWD[X] ∪ FWD[Y],
    ATOM[X] ∪ ATOM[Y],
    root>)
```

Definition 2.20: Let **X** be an SKI-wff. The *graphical transform of* **X** is the SKI-G-wff *STRING-TO-GRAPH*[**X**] where
STRING-TO-GRAPH[**X**] $\overset{\text{Def}}{\rightarrow}$

```
(if ATOM-P[X]
  then ATOMIC-GRAPH[X]
  else ;; X is a combination
   (let opr be STRING-TO-GRAPH[OPERATOR[X]] &
        opd be STRING-TO-GRAPH[OPERAND[X]] in
     ;; opr and opd share no vertices, so they are compatible
     COMBINE[opr,opd]))
```

Incompatible SKI-G-wffs are not COMBINEd as the resulting graph may not be an SKI-G-wff. This is so because the definition of SKI-G-wff does not prevent two SKI-G-wffs from having the same vertex set and inconsistent arc sets at the same time.

For any two composable functions **F** and **G**, **F** $\circ$ **G** represents their composition. For any function **F** capable of being composed with itself, $\mathbf{F}^n$ is the function created by composing **F** with itself n times. That is:

$$\mathbf{F}^n = \mathbf{F} \circ \mathbf{F} \circ \ldots \circ \mathbf{F},$$

where there are n **F**s to the right of the equal sign. $\mathbf{F}^0$ is the identity function.

It can be shown that the graphical transform of an SKI-wff is a clean compact SKI-G-wff without shared subformulas. It can also be shown that, given an SKI-wff **X** and its graphical transform **Y**, the linear transform of **Y** is **X**. That is to say, GRAPH-TO-STRING∘STRING-TO-GRAPH is the identity function on SKI-wffs. It is not the case, however, that STRING-TO-GRAPH∘GRAPH-TO-STRING is the identity function on SKI-G-wffs. Applied to a clean compact SKI-G-wff **Y** having no cycles and no confluences (no shared subformulas), however, an SKI-G-wff **Y′** isomorphic to **Y** is produced. The only difference between **Y** and **Y′** (their graphs will appear identical when displayed) is their vertex sets. As the functions ATOMIC-GRAPH and COMBINE (the functions STRING-TO-GRAPH is defined in terms of) always use *new* vertices, the vertex sets will be necessarily disjoint.

Definition 2.21: Let **X** be an SKI-G-wff. The *initial atom of* **X** is (*INITIAL-ATOM*[**X**]) where:

```
INITIAL-ATOM[X] $\overset{\text{Def}}{\Rightarrow}$
  (let root be ROOT[X] in
    (if FORWARDED-P[root,X]
       then INITIAL-ATOM[SUBFORMULA[X,FORWARDED-TO[root,X]]]
     elseif ATOM-P[X]
       then ATOM[root]
     else ;; X is a combination
       INITIAL-ATOM[OPERATOR[X]]))
```

Definition 2.22: Let **X** be an SKI-G-wff. The *number of arguments of* **X** is (*NUMBER-OF-ARGS*[**X**]) where:

```
NUMBER-OF-ARGS[X] $\overset{\text{Def}}{\Rightarrow}$
  (let root be ROOT[X] in
    (if FORWARDED-P[root,X]
       then NUMBER-OF-ARGS[SUBFORMULA[X,FORWARDED-TO[root,X]]]
     elseif ATOM-P[X]
       then 0
     else ;; X is a combination
       (+ 1 NUMBER-OF-ARGS[OPERATOR[X]])))
```

Definition 2.23: Let **X** be an SKI-G-wff. If $1 \leq n \leq$ NUMBER-OF-ARGS[**X**], then the ***nth argument*** *of* **X** is *ARG*[**n**,**X**] where:

```
ARG[n,X] $\overset{\text{Def}}{\Rightarrow}$
  (let numargs be NUMBER-OF-ARGS[X] in
    $\text{OPERAND} \circ \text{OPERATOR}^{\text{numargs}-\text{n}}[\mathbf{X}]$)
```

2.1.2. Reduction

The "immediately reducible to" relation of the SKI-G-calculus (SKI-G-imr) mirrors the SKI-normal-imr relation on SKI-wffs presented in the preceding chapter. That is to say reduction in the SKI-G-calculus proceeds by contracting the graphical redex corresponding to the SKI-calculus' leftmost SKI-redex. Thus, like the calculus characterized by the set of wffs SKI-wff and "immediately reducible to" relation SKI-normal-imr, the SKI-G-calculus is deterministic.

Some preliminary concepts are presented prior to the definition of SKI-G-imr.

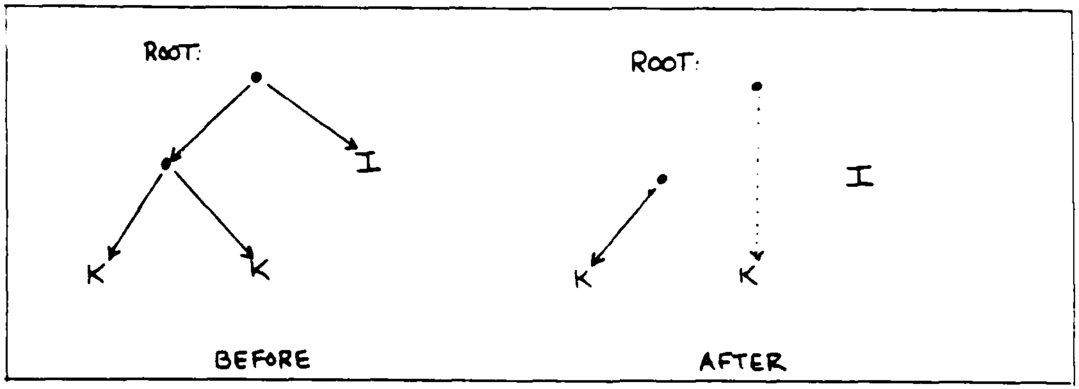
Definition 2.24: Let **X** be a combination whose root is not forwarded. Let **Y** be an SKI-G-wff compatible with (but different from) **X**. The SKI-G-wff which results from forwarding the root of **X** to the root of **Y** is *FORWARD-COMB*[**X**,**Y**] where:
FORWARD-COMB[**X**,**Y**] $\overset{\text{Def}}{=}$

```
(let rootx be ROOT[X] in
  (let rtrx be RATOR[X][rootx] &
      rndx be RAND[X][rootx] in
    <VS[X] ∪ VS[Y],
     RATOR[X] ∪ RATOR[Y] - {<rootx,rtrx>},
     RAND[X] ∪ RAND[Y] - {<rootx,rndx>},
     FWD[X] ∪ FWD[Y] ∪ {<rootx.ROOT[Y]>},
     ATOM[X] ∪ ATOM[Y],
     rootx>))
```

A note on the restriction, in the previous definition, that **Y** must be different from **X**: **Y** cannot be **X** nor can **Y**'s root be forwarded (via one or more arcs) to **X**'s root. Forwarding **X** to such a wff would create a graph which is not an SKI-G-wff.

The reason for merging only compatible wffs is the same as that for COMBINing only compatible wffs — i.e. the graph that results from merging incompatible wffs may not be a wff at all.[2]

Note that combination forwarding makes garbage out of vertices which were previously accessible only from rtrx or rndx.



An Example of Combination Forwarding
Figure 2.8

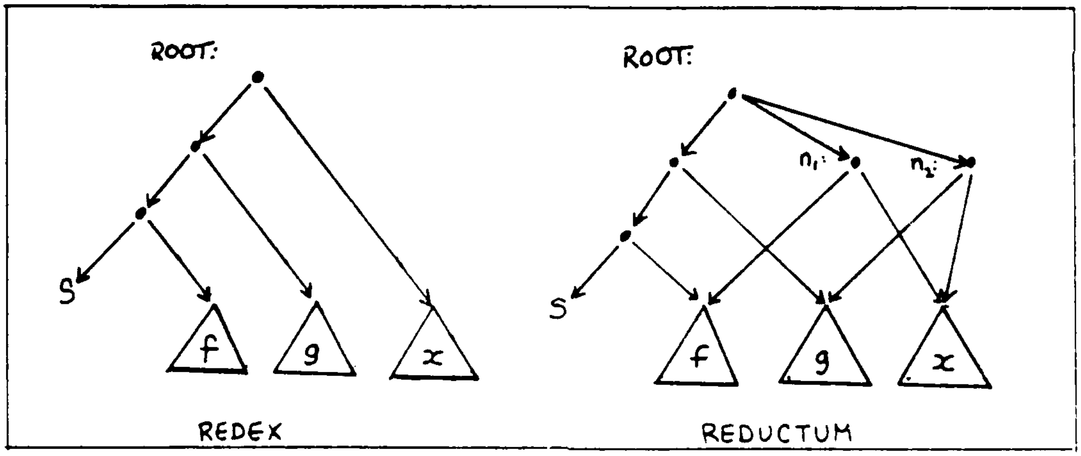
[2] In the implementation, all wffs are compatible. Therefore there is no need to check for compatibility before performing a forwarding operation or before COMBINing two wffs.

Definition 2.25: Let **X** be an SKI-G-wff. **X** is an *SKI-G-S redex* if *SKI-G-S-REDEX-P*[**X**] where:

```
SKI-G-S-REDEX-P[X] $\stackrel{\text{Def}}{=}$
(and (not FORWARDED-P[ROOT[X],X])
    INITIAL-ATOM[X] = S
    NUMBER-OF-ARGS[X] = 3)
```

Definition 2.26: Let **X** be an SKI-G-S redex. The SKI-G-wff **Y** is the *SKI-G-S reductum of* **X** if *SKI-G-S-REDUCTUM*[**X**] = **Y** (**X** *SKI-G-S-imr* **Y**) where:

```
SKI-G-S-REDUCTUM[X] $\stackrel{\text{Def}}{=}$
    (let root be ROOT[X] &
        rf be ROOT[ARG[1,X]] &
        rg be ROOT[ARG[2,X]] &
        rx be ROOT[ARG[3,X]] &
        $nv_1$ be a new vertex &
        $nv_2$ be a new vertex in
       <VS[X] ∪ {$nv_1$,$nv_2$},
        RATOR[X]|(VS[X]-{root}) ∪ {<root,$nv_1$>,<$nv_1$,rf>,<$nv_2$,rg>},
        RAND[X]|(VS[X]-{root}) ∪ {<root,$nv_2$>,<$nv_1$,rx>,<$nv_2$,rx>},
        FWD[X],
        ATOM[X],
        root>)
```



An Example of SKI-G-S Reduction

Figure 2.9

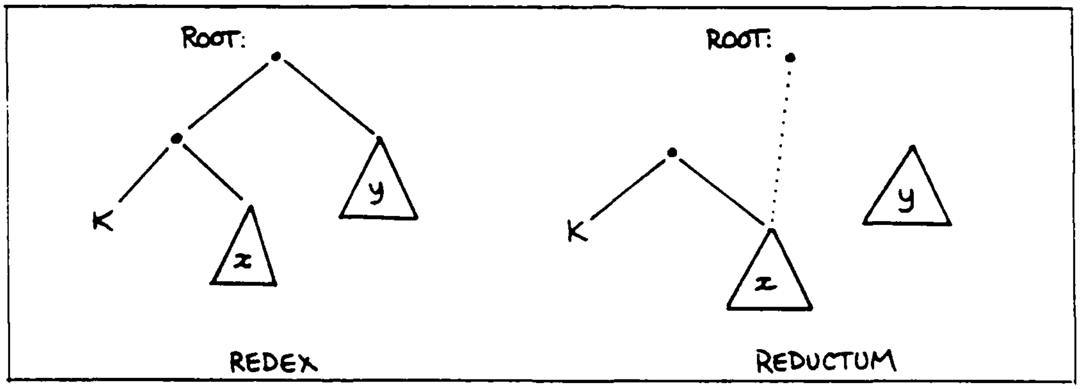
The figure above demands some explanation. The vertices labeled n_1 and n_2 denote the new vertices present in the reductum. The labeled triangles in the above figure (and the figures to follow) represent whole SKI-G-wffs. This representation is a little bit deceiving. These wffs may contain arcs pointing at the other vertices — e.g. the triangle labeled x may contain arcs pointing at vertices in the wff represented by the triangle labeled g (even though no such arcs appear in the representation). Thus, some of the vertices which appear from the figure to be inaccessible from the root may, in fact, be accessible.

Definition 2.27: Let **X** be an SKI-G-wff. **X** is an *SKI-G-K redex* if *SKI-G-K-REDEX-P*[**X**] where:
SKI-G-K-REDEX-P[X] $\overset{\text{Def}}{\rightarrow}$
(*and* (*not* FORWARDED-P[ROOT[X],X])
INITIAL-ATOM[X] = K
NUMBER-OF-ARGS[X] = 2)

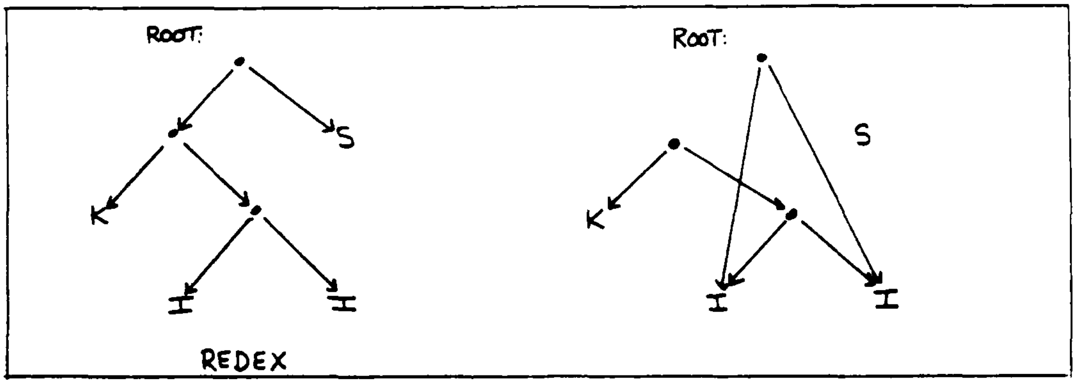
Definition 2.28: Let **X** be an SKI-G-K redex. The SKI-G-wff **Y** is the *SKI-G-K reductum of* **X** if *SKI-G-K-REDUCTUM*[**X**] = **Y** (**X** *SKI-G-K-imr* **Y**) where:
SKI-G-K-REDUCTUM[X] $\overset{\text{Def}}{\rightarrow}$
FORWARD-COMB[X,ARG[1,X]]

The last two definitions are good examples of the close relationship between the definitions of concepts in this formal calculus and the functions which implement them. These definitions can be (almost trivially) realized in most programming languages.

Note that in the definition of SKI-G-K-REDUCTUM, the SKI-G-K redex is forwarded to its first argument. There is a subtle reason for this. One might think that the use of the forwarding pointer could be obviated by simply replacing the RATOR and RAND pointers of the redex with the RATOR and RAND pointers of the first argument. However, if this is done and if the first argument is itself a redex, this replacement would create a duplicate redex. Forming duplicate redexes violates the property of full laziness — that states that every expression is reduced at most once.



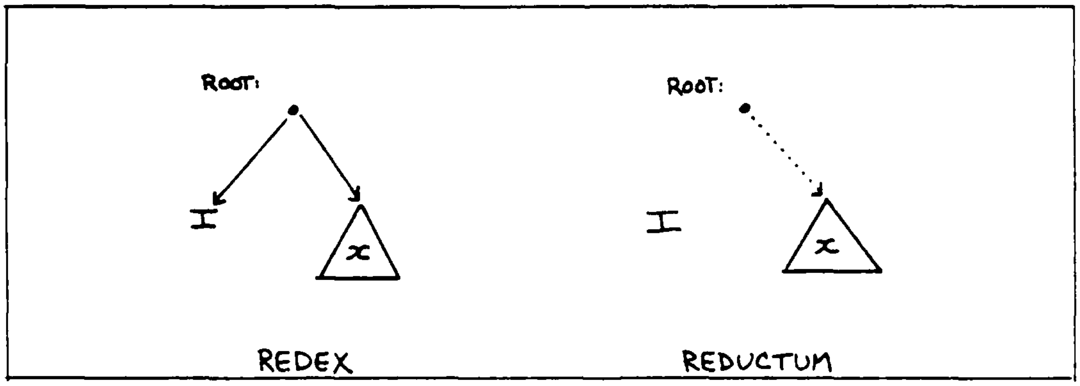
An Example of Proper SKI-G-K Reduction
Figure 2.10



An Example of Improper SKI-G-K Reduction
Figure 2.11

Definition 2.29: Let **X** be an SKI-G-wff. **X** is an *SKI-G-I redex* if *SKI-G-I-REDEX-P*[**X**] where:
SKI-G-I-REDEX-P[X] $\overset{\text{Def}}{\Rightarrow}$
(*and* (*not* FORWARDED-P[ROOT[X],X])
INITIAL-ATOM[X] = I
NUMBER-OF-ARGS[X] = 1)

Definition 2.30: Let **X** be an SKI-G-I redex. The SKI-G-wff **Y** is the *SKI-G-I reductum of* **X** if *SKI-G-I-REDUCTUM*[**X**] = **Y**, (**X** *SKI-G-I-imr* **Y**) where:
SKI-G-I-REDUCTUM[X] $\overset{\text{Def}}{\Rightarrow}$
FORWARD-COMB[X,ARG[1,X]]



An Example of SKI-G-I Reduction
Figure 2.12

Note that if **X** R **Y**, where R is either SKI-G-S-imr, SKI-G-K-imr, or SKI-G-I-imr, then VS[**Y**] contains VS[**X**]. Reductions do not discard vertices.

It is often convenient, just as with SKI-G-wffs, to express the relations SKI-G-S-imr, SKI-G-K-imr, and SKI-G-I-imr linearly. Written in this manner, they are, respectively:

S **X Y Z** → **X Z** (**Y Z**)
K **X Y** → **X**
I **X** → **X**

Of course, the relations, expressed linearly, are subject to the same problems as are linear representations of SKI-G-wffs:

- Shared subformulas appear as duplicate subformulas (e.g. in the S reduction rule, the wffs denoted by **Z** are actually the same wff)
- Forwarding arcs are invisible (e.g. in the K and I reduction rules, the root of the wff denoted by **X** is a forwarding vertex)

These relations, like their linear counterparts in the SKI-calculus, are also often referred to as substantive reduction rules, as each specifies a redex-reductum pair.

Definition 2.31: Each functor has an *arity* determined by its reduction rule. The arity of a functor **f** (*ARITY*[**f**]) having reduction rule: $\mathbf{f}\ \mathbf{X}_1 \cdots \mathbf{X}_n \rightarrow \mathbf{Z}$ is n . S, therefore, has arity 3, K has arity 2, and I has arity 1.

In the LNF-calculus, some functors are characterized by more than one reduction rule. These functors' rules, however, always require the same number of arguments. Thus such a functor's arity may be determined by examining any one of its rules.

Hereafter, for conciseness (in contexts in which no confusion will arise) the "SKI-G-" prefix may be dropped from such identifiers as: SKI-G-wff, SKI-G-S-REDEX-P, SKI-G-K-imr, etc.

Definition 2.32: Let **X** be an SKI-G-wff. **X** is an *SKI-G redex* iff *SKI-G-REDEX-P*[**X**] where
SKI-G-REDEX-P[**X**] $\overset{\text{Def}}{\rightarrow}$
(*or* S-REDEX-P[**X**] K-REDEX-P[**X**] I-REDEX-P[**X**])

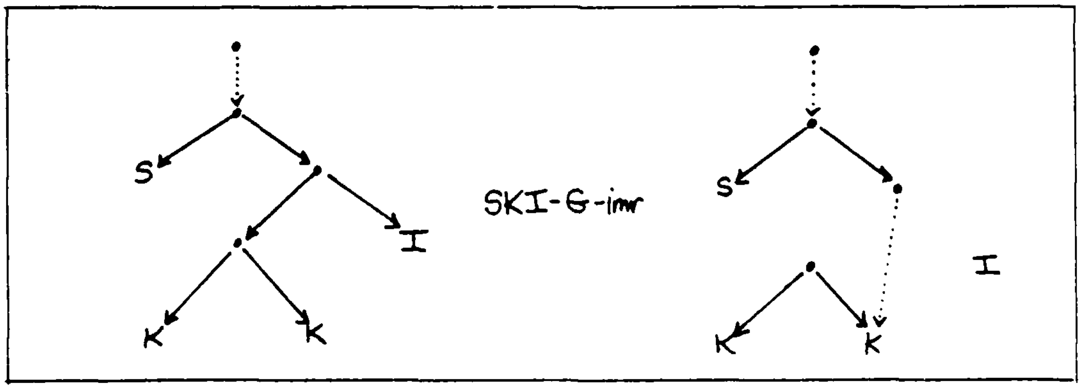
Definition 2.33: Let **X** be an SKI-G-wff. **X** *contains an initial redex* iff
(*or* SKI-G-REDEX-P[**X**]
OPERATOR[**X**] contains an initial redex)

Definition 2.34: Let **X** be an SKI-G-wff. **X** is in *SKI-G-lazy-normal form* iff *SKI-G-LAZY-NF-P*[**X**] where
SKI-G-LAZY-NF-P[**X**] $\overset{\text{Def}}{\rightarrow}$
X does not contain an initial redex

The definition of SKI-G-imr (next) is a bit long and complicated. It is complicated by the presence of forwarding pointers and the fact that, because of shared subformulas and cycles in the wff, redex contractions can be a bit more difficult to formalize than in a string oriented calculus. However, the informal description of the relation is quite simple to comprehend. Informally, an SKI-G-wff **X** reduces immediately to **Y** iff either <**X**,**Y**> is a redex-reductum pair or **X** contains a leftmost redex and **Y** is the wff which results from contracting this redex.

Definition 2.35: Given SKI-G-wffs **X** and **Y**. **X** *immediately reduces to* **Y** iff **X** *SKI-G-imr* **Y** where

```
X SKI-G-imr Y $\overset{Def}{\rightarrow}$
  (let xroot be ROOT[X] in
   (if FORWARDED-P[xroot,X]
    then (let yroot be ROOT[Y] in
          (and FORWARDED-P[yroot,Y]
               xroot = yroot
               (SUBFORMULA[X,FORWARDED-TO[xroot,X]]
                SKI-G-imr
                SUBFORMULA[Y,FORWARDED-TO[yroot,Y]])))
    elseif (not LAZY-NF-P[X])
     then (or X S-imr Y
              X K-imr Y
              X I-imr Y
              (and COMBINATION-P[X]
                   (there is a $Y_{OPR}$ ∈ SKI-G-wff such that
                     (and OPERATOR[X] SKI-G-imr $Y_{OPR}$
                          Y = SKI-G-WFF[$Y_{OPR}$,xroot])))
    else ;; X does not contain an initial redex
     (and COMBINATION-P[X]
          (there is an i ∈ 1,...,NUM-ARGS[X]
           and an SKI-G-wff $Y_{ARG_i}$ such that
          (and ARG[i,X] SKI-G-imr $Y_{ARG_i}$
               Y = SKI-G-WFF[$Y_{ARG_i}$,xroot]
               there isn't a j ∈ 1,...,i-1 such that
                 ARG[j,X] is reducible))))
```



An Example of SKI-G Reduction
Figure 2.13

Definition 2.36: *SKI-G-red* is the transitive closure of SKI-G-imr.

Definition 2.37: *SKI-G-red** is the reflexive transitive closure of SKI-G-imr.

Definition 2.38: Let **X** be an SKI-G-wff. **X** is in *SKI-G-normal form* iff *SKI-G-NF-P*[**X**] where

SKI-G-NF-P[**X**] $\overset{\text{Def}}{=}$

no subformula of **X** is an SKI-G-REDEX

Definition 2.39: Let **X** be an SKI-G-wff which is not in SKI-G-normal form. The *leftmost redex of* **X** is *LEFTMOST-REDEX*[**X**] where:

```
LEFTMOST-REDEX[X] Def=
  (if REDEX-P[X]
    then X
   elseif OPERATOR[X] contains a redex
    then LEFTMOST-REDEX[OPERATOR[X]]
   else ;; OPERAND[X] contains a redex
    LEFTMOST-REDEX[OPERAND[X]])
```

Note that the SKI-G-calculus is deterministic. For any SKI-G-wff **X**, there is only one reduction sequence starting at **X**. This is true because each reduction step involves contracting the wff's leftmost redex, which (if it exists) is unique. Moreover, if **X** has an SKI-G-normal form, then it is arrived at by first being reduced to SKI-G-lazy-normal form. Each argument, in turn, is then reduced to SKI-G-normal form.

The following results show that any SKI-calculus reduction sequence[3] (and therefore any λ-calculus reduction sequence[4]) can be simulated by a reduction sequence (often involving fewer reductions) in the SKI-G-calculus. These results also demonstrate that any SKI-G-calculus reduction can be simulated in the SKI-calculus. Thus, the SKI-G-calculus is shown to be equivalent in power to the SKI-calculus, the λ-calculus, et al.

Lemma 2.1: Let **SKI-X** be a variable-free SKI-wff. If **SKI-X** SKI-normal-imr **SKI-Y**, then there is an **SKI-G-Y** $\in$ SKI-G-wff such that STRING-TO-GRAPH[**SKI-X**] SKI-G-imr **SKI-G-Y** and **SKI-Y** = GRAPH-TO-STRING[**SKI-G-Y**].

Proof Sketch:

STRING-TO-GRAPH preserves redexes. Thus, the leftmost redex in **SKI-X**, which when contracted yields **SKI-Y**, will have a counterpart in STRING-TO-GRAPH[**SKI-X**] (**SKI-G-X**) which will also be a leftmost redex. Contracting this redex, yielding **SKI-G-Y**, will have no effect on the rest of the graph **SKI-G-X** as it does not contain any confluences or cycles. Thus, since the redex-reductum pairs of the SKI-G-calculus mirror the redex-reductum pairs in the SKI-calculus, the string transform of **SKI-G-Y** will be **SKI-Y**.

End Sketch

The previous lemma demonstrates that a single reduction step in the SKI-calculus can be simulated by a single reduction step in the SKI-G-calculus. The next lemma states that a single reduction step in the SKI-G-calculus can be simulated by one *or more* reduction steps in the SKI-calculus.

[3] with the restriction that the initial SKI-wff in the sequence does not contain any variables

[4] with the restriction that the initial λ-wff is closed

Lemma 2.2: Let **SKI-G-X** be an SKI-G-wff. If **SKI-G-X** SKI-G-imr **SKI-G-Y**, then GRAPH-TO-STRING[**SKI-G-X**] SKI-red GRAPH-TO-STRING[**SKI-G-Y**].

Proof Sketch:

The SKI-wff GRAPH-TO-STRING[**SKI-G-X**] (**SKI-X**) contains N copies of each subformula of **SKI-G-X** having N distinct paths from **SKI-G-X**'s root to the root of the subformula. In particular, if there are M distinct paths from **SKI-G-X**'s root to the root of the redex contracted, then the SKI-wff **SKI-X** contains M copies of this redex. Each of these M redexes must be contracted as the SKI-wff GRAPH-TO-STRING[**SKI-G-Y**] (**SKI-Y**) will contain M copies of the redex's reductum. The SKI-wffs **SKI-X** and **SKI-Y** will therefore stand in the relation SKI-red if these M redexes are all contracted.

End Sketch

The following two conjectures claim equivalence between the SKI-calculus and the SKI-G-calculus.

Conjecture 2.1: Let **SKI-X** be a variable-free SKI-wff. If **SKI-X** SKI-normal-red **SKI-Y**, where **SKI-Y** in SKI-normal form, then there is an **SKI-G-Y** ∈ SKI-G-wff in SKI-G-normal form such that STRING-TO-GRAPH[**SKI-X**] SKI-G-red **SKI-G-Y** and **SKI-Y** = GRAPH-TO-STRING[**SKI-G-Y**].

Proof Sketch:

The reduction sequence in the SKI-G-calculus would mirror the reduction sequence in the SKI-calculus with the following exception. In an SKI-calculus reduction step redexes are often copied (e.g. any redex in **Z** after the step: S **X Y Z** → **X Z** (**Y Z**)). On the other hand, redexes are never duplicated in an SKI-G-calculus reduction. Thus, the SKI-G-calculus reduction sequence may be shorter than the one in the SKI-calculus — how much shorter depends, of course, on how many redexes are copied in the SKI-calculus reduction sequence. Note the requirement in the theorem statement that the SKI-reduction sequence terminates in an SKI-wff in SKI-normal form. Some reduction sequences which do not eliminate all redexes cannot be simulated — those which fail to contract the redexes they copy.

End Sketch

It has been informally argued that any SKI-normal reduction sequence resulting in an SKI-wff in SKI-normal form can be simulated in the SKI-G-calculus. It remains to show that all SKI-G-calculus reductions can be simulated in the SKI-calculus.

Conjecture 2.2: Let **SKI-G-X** be an acyclic SKI-G-wff. If **SKI-G-X** SKI-G-red **SKI-G-Y**, then GRAPH-TO-STRING[**SKI-G-X**] SKI-red GRAPH-TO-STRING[**SKI-G-Y**].

Proof Sketch:

The SKI-calculus reduction sequence which simulates the SKI-G-calculus reduction sequence will mirror the graph sequence except that it may take more steps (to reduce the copies of the redexes it has created). There need not be a requirement that the simulated sequence end in a redex-free graph as no copies of redexes are created by it.

End Sketch

2.1.3. On Realizing the SKI-G-calculus

Since any closed λ-wff can be translated into an SKI-G-wff, reduced, and then transformed back, an SKI-G machine (one which produces SKI-G-wffs in SKI-G-normal form from arbitrary SKI-G-wffs it has been provided) could be used as the reduction engine at the core of a functional programming language implementation. This machine (SKI-G-M) could be built from a simpler machine (LNF-M) which accepts SKI-G-wffs as input and produces SKI-G-wffs in SKI-G-*lazy*-normal form. An informal definition of SKI-G-M in terms of LNF-M follows:

$$\begin{array}{l} \text{SKI-G-M}[\mathbf{X}] \overset{\text{Def}}{=} \\ \quad (\textit{let}\ \mathbf{a}\ \mathrm{E}_1 \cdots \mathrm{E}_n\ \textit{be}\ \text{LNF-M}[\mathbf{X}]\ \textit{in} \\ \qquad \mathbf{a}\ \text{SKI-G-M}[\mathrm{E}_1] \ \cdots\ \text{SKI-G-M}[\mathrm{E}_n\,]) \end{array}$$

Besides being an elegant machine architecture, it has two properties which make it an efficient one as well. Firstly, the only redexes contracted are initial redexes. These redexes are easy to locate within a wff as only the "left spine" of the graph need be searched. Secondly, having reduced the input wff to lazy-normal form, the structure of the output wff is known — that is, both its initial atom and number of arguments are known. Further reductions of the wff only affect the structure of the wff's arguments. Thus, having reached lazy-normal form, the initial atom may be output and reduction started on the arguments.

However, basing the implementation of a usable[5] functional programming (FP) language on SKI-G-M (the architecture notwithstanding) is problematic. The two most significant problems with this approach are:

1. All of the constructs (both in data: like numbers and lists, and in code: like conditional expressions and expressions with auxiliary declarations) programmers have become accustomed to, and now expect to find in an FP language, must be represented by SKI-G-wffs.[6]
2. Translating complex (closed) λ-wffs into SKI-G-wffs creates SKI-G-wffs of unacceptable size. The translated SKI-G-wff grows exponentially with the number of nested abstractions present in the λ-wff ([Turner 1979c]).

Assuming that the FP language to be implemented is a "sugared" version of the λ-calculus, the desugaring process must represent all of the constructs of the language as λ-wffs. For example, natural numbers are data items most programmers would expect to find in an FP language. These numbers must be represented as λ-wffs. Although this can be done, the resulting wffs are large in size and difficult to manipulate. The arithmetic operators must be coded as λ-wffs as well. Besides being complex, the desugared expressions (now λ-wffs) have lost something in the process. One cannot distinguish a desugared numeral from a function — the programmer's intention has been lost. Any NUMBER-P predicate, for example, would return TRUE[7] when provided with any λ-wff taking the form of a natural number representation, even though that was not its

[5] capable of running more than the customary set of trivial test programs, not necessarily a production quality system

[6] This is not just a problem with the SKI-G-calculus, of course — all of the other calculi previously presented also suffer from this malady

[7] a λ-wff representation of TRUE

intended use.

The representation problem discussed above, by increasing the size and complexity of the λ-wffs (which then must be translated into SKI-G-wffs), makes the second problem even more significant.

The LNF-calculus is a directly realizable version of the SKI-G-calculus. The LNF-calculus, as the reader will see, does not possess either of the problems which prevent the SKI-G-calculus from being the basis of an efficient programming system.

2.2. The LNF-calculus

As mentioned several times, the LNF-calculus is the reduction calculus which has been realized in ZetaLisp on a Lisp machine. This realization is the reduction engine of the FP language LNF.

Detailed in this section are the modifications made to the SKI-G-calculus which transform it into the LNF-calculus. The resulting formal system is one that has been directly implemented resulting in a usable FP system. The implementation, described in detail in Chapter 3, mirrors the definition of the LNF-calculus to follow.

2.2.1. Constructions, Functions, and Unknowns

LNF-wffs, like SKI-wffs and SKI-G-wffs, are either atoms or combinations. Combinations are composite wffs, having an operator and an operand, both of which are LNF-wffs.

The definition of LNF-wff is identical to that of SKI-G-wff except for the clause:

> **ATOM** is a nonempty partial function from **VS** to {S,K,I}.

In the definition of LNF-wff, this clause is replaced with:

> **ATOM** is a nonempty partial function from **VS** to LNF-FUNCT ∪ LNF-CONS.

Definition 2.40: *LNF-FUNCT* is the LNF-calculus' set of functors and *LNF-CONS* is the LNF-calculus' set of constructors. LNF-FUNCT and LNF-CONS partition the set of identifiers. An identifier is in LNF-FUNCT iff it is associated with a reduction rule. All other identifiers are in LNF-CONS.

Definition 2.41: Let **X** be an LNF-wff. If **X** has initial atom **a** and **a** is a constructor, then **X** is a *construction* (*CONSTRUCTION-P*[**X**]). If **X** has initial atom **a** and **a** is a functor and NUMBER-OF-ARGS[**X**] < ARITY[**a**], then **X** is a *function* (*FUNCTION-P*[**X**]). If **X** is neither a construction nor a function, then it is an *unknown* (*UNKNOWN-P*[**X**]).

Hence, all LNF-wffs are either constructions, functions, or unknowns.

Henceforth, metavariables denoting LNF-wffs will come in different flavors. The following table summarizes these new conventions:[8]

Metavariable	LNF-wff Class
A,B,W,X,Y,Z	LNF-wff of any type
f	Functor
FN	Function
c	Constructor
CN	Construction
cf	Constructor or functor
CFN	Construction or function
RDU	Reducible unknown
IMR	In expressions containing **RDU**, the wff s.t. **RDU** LNF-imr **IMR**
IRU	Irreducible unknown
b	TRUE or FALSE
i,j	Integer
s,t	Floating point number
n,m,o	Floating point number or integer
P	Pair (a wff having the linear representation: PAIR **X Y**)

LNF-wff Metavariables

Some examples of linear representations of LNF-wffs using this new notation follow:

c n IRU	a construction whose first argument is a number and whose second argument is an irreducible unknown
$\mathbf{c}\ \mathbf{X}_1 \cdots \mathbf{X}_k$	a construction having k arguments
+ **RDU CFN**	an LNF-wff having initial atom +, a reducible unknown as first argument, and a construction or function as second argument
f (**c Z**)	an LNF-wff having a functor as initial atom and a first argument which is a construction having one argument

In the following sections, the new functors (and their associated reduction rules) will be presented. The first functors to be presented will be those defined by H.B. Curry and D.A. Turner.

2.2.2. Curry's and Turner's Functors

When translating (closed) λ-wffs to SKI-G-wffs, the most significant problem is that the size of the SKI-G-wff grows exponentially with the number of nested abstractions in the λ-wff. This problem is diminished by introducing several new functors and modifying Schönfinkel's ABSTRACTion algorithm to make use of them.

H.B. Curry, in [Curry 1958], introduced three new functors (B, C, and W) and a modified ABSTRACTion algorithm. With this new algorithm, the translated λ-wffs did

[8] These new metavariables may appear decorated with subscripts as well.

not grow as rapidly. D.A. Turner claims, in [Turner 1979c], that the growth rate is still at least quadratic in the number of variables abstracted. D.A. Turner, in [Turner 1979c] and [Turner 1984], modified the algorithm further by adding three more functors, similar to S, B, and C, which he called S′, B′, and C′. Although in the worst case the translated λ-wffs still may grow quadratically, in practice most λ-wffs only grow linearly when translated. Formal results concerning the growth of translated λ-wffs can be found in [Kennaway 1982] and [Burton 1982]. Kennaway proves that the wff which results from this transformation grows, in the worst case, at a rate proportional to the square of the size of the original λ-wff. Burton gives an algorithm which balances wffs — unbalanced wffs are the ones which give rise to the quadratric growth. The resulting balanced wffs grow, when their variables are removed, at only a linear rate. Burton's algorithm, however, is restricted to λ-wffs in which no abstractions contain global variables. He claims that any λ-wff may be transformed into a λ-wff having this property — but at the cost of (in the worst case) quadratic growth!

In order to construct LNF-wffs from closed λ-wffs, one could use a modified Schönfinkel algorithm which produces strings and then use the STRING-TO-GRAPH function to produce LNF-wffs. Presented below are two functions which together transform (closed) λ-wffs directly into LNF-wffs by employing the new functors defined by Curry and Turner.

Definition 2.42: Let **X** be a λ-wff. The *LNF transform of* **X** is λ-*TO-LNF*[X] where:

```
λ-TO-LNF[X] $\overset{Def}{=}$
  (if ATOM-P[X]
    then ATOMIC-GRAPH[X]
  (elseif X = (λ v B)
    then C-T-ABS[v,λ-TO-LNF[B]]
   else ;; X is a combination
    (let OPR be λ-TO-LNF[OPERATOR[X]] &
         OPD be λ-TO-LNF[OPERAND[X]] in
      ;; OPR and OPD share no vertices so they are compatible
      COMBINE[OPR,OPD]))
```

In the following definition of the Schönfinkel-Curry-Turner-Scheevel abstraction algorithm (C-T-ABS), the shorthand notation:

E of the form: B **X Y**

replaces the rather cumbersome phrase "E is a combination having initial atom B and two arguments — let the first of which be called **X** and the second **Y**.

Definition 2.43: For any variable v and LNF-wff **B**, there is an LNF-wff *C-T-ABS*[v,**B**] where

```
C-T-ABS[v,B] $\overset{\text{Def}}{\Rightarrow}$
  (if (and ATOM-P[B]  INITIAL-ATOM[B] = v)
     then ATOMIC-GRAPH[I]
    elseif v does not occur in B
     then K-COMB[B]
    else ;; B is a combination
     (let OPR be OPERATOR[B] &
          OPD be OPERAND[B] in
        (if (and ATOM-P[OPD]  INITIAL-ATOM[OPD] = v)
          then (if v does not occur in OPR
                  then OPR
                else W-COMB[C-T-ABS[v,OPR]])
         elseif v occurs in both OPR and OPD
          then (let ABS-OPR be C-T-ABS[v,OPR] in
                 (if ABS-OPR of the form: B X Y
                    then S'-COMB[X,Y,C-T-ABS[v,OPD]]
                   else S-COMB[ABS-OPR,C-T-ABS[v,OPD]]))
         elseif v occurs in OPD ;; but not in OPR
          then (let ABS-OPD be C-T-ABS[v,OPD] in
                 (if ABS-OPD of the form: B X Y
                    then B'-COMB[OPR,X,Y]
                   else B-COMB[OPR,ABS-OPD]))
         else ;; v occurs in OPR but not in OPD
          (let ABS-OPR be C-T-ABS[v,OPR] in
            (if ABS-OPR of the form: B X Y
               then C'-COMB[X,Y,OPD]
             else C-COMB[ABS-OPR,OPD])))
```

Some auxiliary definitions of functions used above:

Definition 2.44: Let **X**, **Y**, and **Z** be LNF-wffs.

K-COMB[X] $\overset{\text{Def}}{\Rightarrow}$
COMBINE[ATOMIC-GRAPH[K],X]

W-COMB[X] $\overset{\text{Def}}{\Rightarrow}$
COMBINE[ATOMIC-GRAPH[W],X]

S'-COMB[X,Y,Z] $\overset{\text{Def}}{\Rightarrow}$
COMBINE[COMBINE[COMBINE[ATOMIC-GRAPH[S'],X],Y],Z]

S-COMB[X,Y] $\overset{\text{Def}}{\Rightarrow}$
COMBINE[COMBINE[ATOMIC-GRAPH[S],X],Y]

B'-COMB[X,Y,Z] $\overset{\text{Def}}{\Rightarrow}$
COMBINE[COMBINE[COMBINE[ATOMIC-GRAPH[B'],X],Y],Z]

B-COMB[X,Y] $\overset{\text{Def}}{\Rightarrow}$
COMBINE[COMBINE[ATOMIC-GRAPH[B],X],Y]

C'-COMB[X,Y,Z] $\overset{\text{Def}}{\Rightarrow}$
COMBINE[COMBINE[COMBINE[ATOMIC-GRAPH[C'],X],Y],Z]

C-COMB[X,Y] $\overset{\text{Def}}{\Rightarrow}$
COMBINE[COMBINE[ATOMIC-GRAPH[C],X],Y]

It is claimed that the wff C-T-ABS[v,B] is equivalent to the wff ABSTRACT[v,B]. They are equivalent in the sense that both (GRAPH-TO-STRING[C-T-ABS[v,B]] Z) and (ABSTRACT[v,B] **Z**), for all SKI-wffs **Z**, reduce to the same SKI-wff **W** given the extended definition of reduction below. An informal justification of the claim follows this definition.

First, recall the reduction rules for S, K, and I:

S **X Y Z** → **X Z** (**Y Z**)
K **X Y** → **X**
I **X** → **X**

Add to these the reduction rules for W, B, C, S', B', and C':

W **X Y** → **X Y Y**
B **X Y Z** → **X** (**Y Z**)
C **X Y Z** → **X Z Y**
S' **W X Y Z** → **W** (**X Z**) (**Y Z**)
B' **W X Y Z** → **W** (**X** (**Y Z**))
C' **W X Y Z** → **W** (**X Z**) **Y**

Turner, in [Turner 1984], gives credit to Mark Scheevel (Burroughs Corporation) for coming up with the B' functor described herein. Turner's B', defined in [Turner 1979c], was defined by this reduction rule: B' **W X Y Z** → **W X** (**Y Z**).

This set of rules, together with the SKI-calculus' contextual reduction rules, defines an extended "immediately reducible to" relation — call it SKI′-imr. The reflexive transitive closure of SKI′-imr is the new reduction relation.

To demonstrate the claimed equivalence, the definition of C-T-ABS is viewed as a collection of rules of the form: <condition> ⇒<wff>. The conditions are enumerated below. Following each condition is the wff (GRAPH-TO-STRING[C-T-ABS[v,B]] Z), a reduction of it, the wff (ABSTRACT[v,B] Z), and a reduction of it. Both reduction sequences end in equivalent wffs.

1. **B** = **v** ⇒
   C-T-ABS : I **Z**
   ABSTRACT: I **Z** ✓

2. **v** doesn't occur in **B** ⇒
   C-T-ABS : K **B** **Z**
   ABSTRACT: K **B** **Z** ✓

3. **B** is a combination, **v** = **B**'s operand, and **v** doesn't occur in **B**'s operator ⇒
   C-T-ABS : OPERATOR[**B**] **Z**
   ABSTRACT: S (K OPERATOR[**B**]) I **Z**
   → K OPERATOR[**B**] **Z** (I **Z**)
   → OPERATOR[**B**] (I **Z**)
   → OPERATOR[**B**] **Z** ✓

4. **B** is a combination, **v** = **B**'s operand, and **v** occurs in **B**'s operator ⇒
   C-T-ABS : W C-T-ABS[**v**,OPERATOR[**B**]] **Z**
   → C-T-ABS[**v**,OPERATOR[**B**]] **Z** **Z**
   ABSTRACT: S ABSTRACT[**v**,OPERATOR[**B**]] I **Z**
   → ABSTRACT[**v**,OPERATOR[**B**]] **Z** (I **Z**)
   → ABSTRACT[**v**,OPERATOR[**B**]] **Z** **Z** ✓

5. **B** is a combination, **v** occurs in **B**'s operator and operand, **v** ≠ **B**'s operand, and C-T-ABS[**v**,OPERATOR[**B**]] = (B **X** **Y**) ⇒
   C-T-ABS : S′ **X** **Y** C-T-ABS[**v**,OPERAND[**B**]] **Z** where
   Y = C-T-ABS[**v**,OPERAND[OPERATOR[**B**]]]
   → **X** (**Y** **Z**) (C-T-ABS[**v**,OPERAND[**B**]] **Z**)
   ABSTRACT: S (S (K **X**) **Y′**) ABSTRACT[**v**,OPERAND[**B**]] **Z** where
   Y′ = ABSTRACT[**v**,OPERAND[OPERATOR[**B**]]]
   → S (K **X**) **Y′** **Z** (ABSTRACT[**v**,OPERAND[**B**]] **Z**)
   → K **X** **Z** (**Y′** **Z**) (ABSTRACT[**v**,OPERAND[**B**]] **Z**)
   → **X** (**Y′** **Z**) (ABSTRACT[**v**,OPERAND[**B**]] **Z**) ✓

6. **B** is a combination, **v** occurs in **B**'s operator and operand, **v** ≠ **B**'s operand, and C-T-ABS[**v**,OPERATOR[**B**]] ≠ (B **X** **Y**) ⇒
   C-T-ABS : S C-T-ABS[**v**,OPERATOR[**B**]] C-T-ABS[**v**,OPERAND[**B**]] **Z**
   ABSTRACT: S ABSTRACT[**v**,OPERATOR[**B**]] ABSTRACT[**v**,OPERAND[**B**]] **Z**
   ✓

7. **B** is a combination, **v** occurs in **B**'s operand but not in **B**'s operator, and C-T-ABS[**v**,OPERAND[**B**]] = (B **X Y**) ⇒
   C-T-ABS : B′ OPERATOR[**B**] **X Y Z** where
   Y = C-T-ABS[**v**,OPERAND[OPERAND[**B**]]]
   → OPERATOR[**B**] (**X** (**Y Z**))
   ABSTRACT: S (K OPERATOR[**B**]) (S (K **X**) **Y**′) **Z** where
   Y′ = ABSTRACT[**v**,OPERAND[OPERAND[**B**]]]
   → K OPERATOR[**B**] **Z** (S (K **X**) **Y**′ **Z**)
   → OPERATOR[**B**] (S (K **X**) **Y**′ **Z**)
   → OPERATOR[**B**] (K **X Z** (**Y**′ **Z**))
   → OPERATOR[**B**] (**X** (**Y**′ **Z**)) √

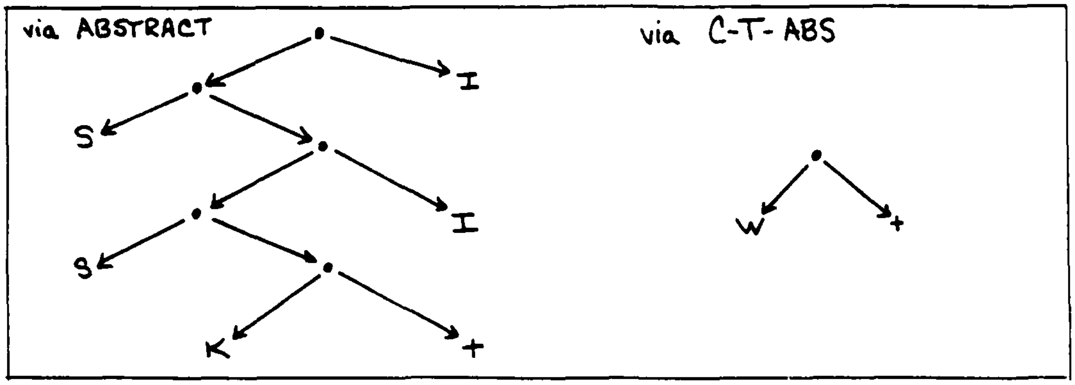
8. **B** is a combination, **v** occurs in **B**'s operand but not in **B**'s operator, and C-T-ABS[**v**,OPERAND[**B**]] ≠ (B **X Y**) ⇒
   C-T-ABS : B OPERATOR[**B**] C-T-ABS[**v**,OPERAND[**B**]] **Z**
   → OPERATOR[**B**] (C-T-ABS[**v**,OPERAND[**B**]] **Z**)
   ABSTRACT: S (K OPERATOR[**B**]) ABSTRACT[**v**,OPERAND[**B**]] **Z**
   → K OPERATOR[**B**] **Z** (ABSTRACT[**v**,OPERAND[**B**]] **Z**)
   → OPERATOR[**B**] (ABSTRACT[**v**,OPERAND[**B**]] **Z**) √

9. **B** is a combination, **v** occurs in **B**'s operator but not in **B**'s operand, and C-T-ABS[**v**,OPERATOR[**B**]] = (B **X Y**) ⇒
   C-T-ABS : C′ **X Y** OPERAND[**B**] **Z** where
   Y = C-T-ABS[**v**,OPERAND[OPERATOR[**B**]]]
   → **X** (**Y Z**) OPERAND[**B**]
   ABSTRACT: S (S (K **X**) **Y**′) (K OPERAND[**B**]) **Z** where
   Y′ = ABSTRACT[**v**,OPERAND[OPERATOR[**B**]]]
   → S (K **X**) **Y**′ **Z** (K OPERAND[**B**] **Z**)
   → K **X Z** (**Y**′ **Z**) (K OPERAND[**B**] **Z**)
   → **X** (**Y**′ **Z**) (K OPERAND[**B**] **Z**)
   → **X** (**Y**′ **Z**) OPERAND[**B**] √

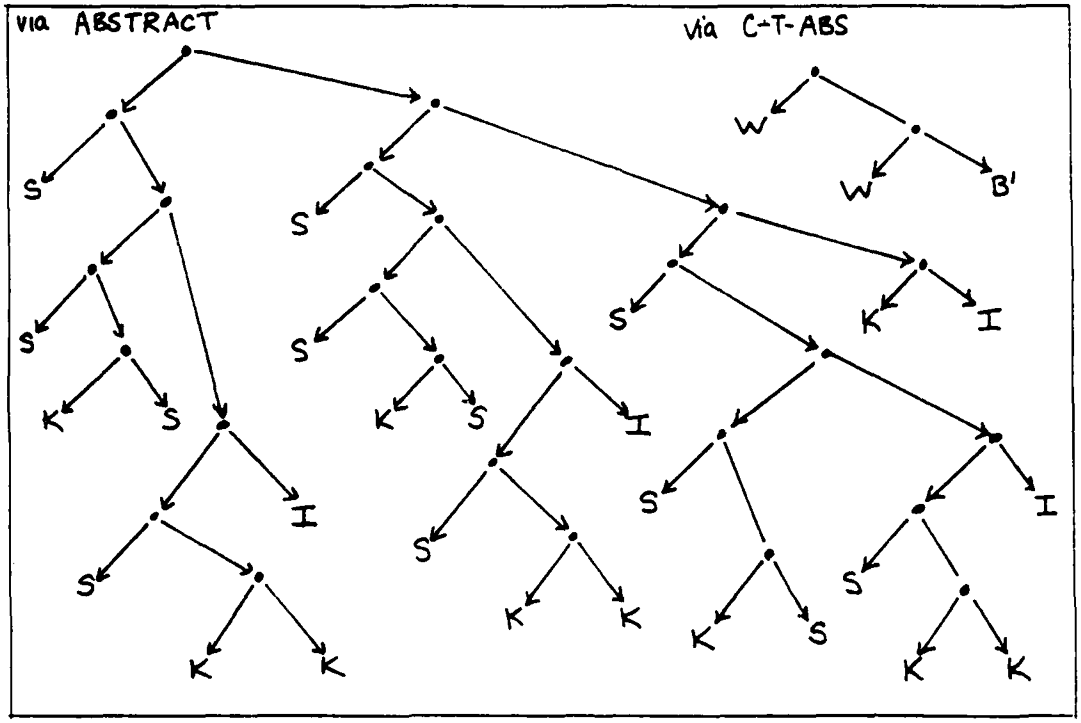
10. **B** is a combination, **v** occurs in **B**'s operator but not in **B**'s operand, and C-T-ABS[**v**,OPERATOR[**B**]] ≠ (B **X Y**) ⇒
    C-T-ABS : C C-T-ABS[**v**,OPERATOR[**B**]] OPERAND[**B**] **Z**
    → C-T-ABS[**v**,OPERATOR[**B**]] **Z** OPERAND[**B**]
    ABSTRACT: S ABSTRACT[**v**,OPERATOR[**B**]] (K OPERAND[**B**])
    → ABSTRACT[**v**,OPERATOR[**B**]] **Z** (K OPERAND[**B**] **Z**)
    → ABSTRACT[**v**,OPERATOR[**B**]] **Z** OPERAND[**B**] √

Note that the third clause in C-T-ABS's definition is very important. This clause actually shrinks the size of the output wff. This transformation step is valid since, in the λ-calculus, the λ-wff ((λ **v** (**X v**)) **Y**) λ-imr (**X Y**) for all λ-wffs **X** and **Y** given that **v** does not occur free in **X**.

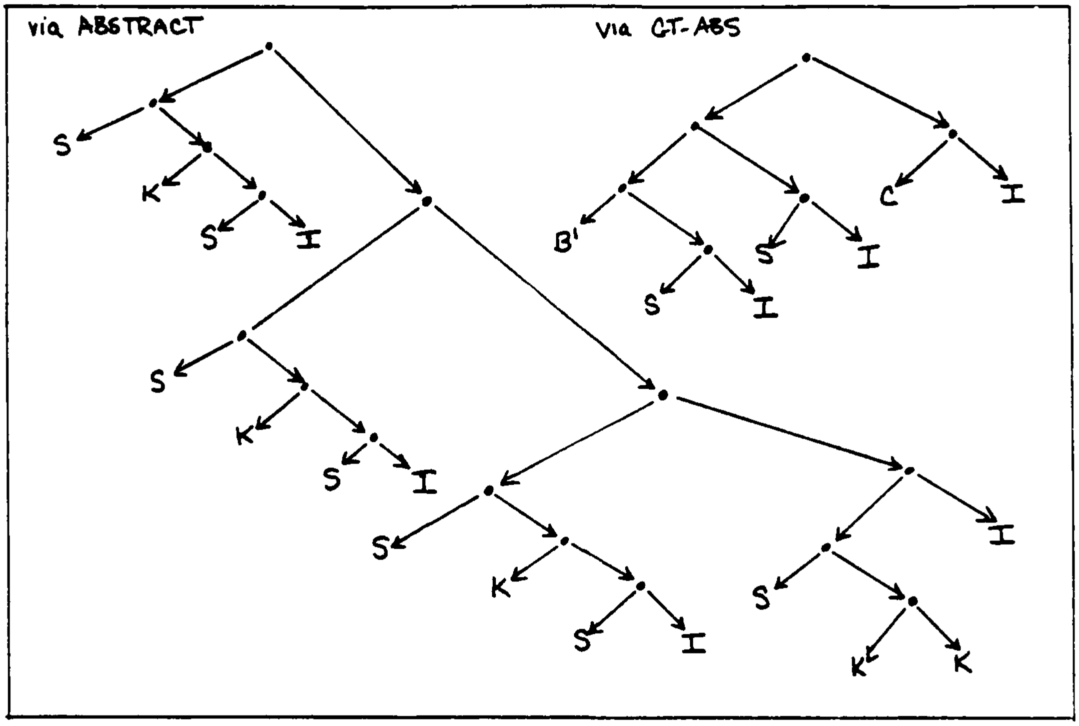
Some examples of λ-wffs and their LNF-wff equivalents (via λ-TO-LNF and C-T-ABS) are displayed below. For comparison, the λ-wffs are also transformed to LNF-wffs via λ-TO-SKI, ABSTRACT, and STRING-TO-GRAPH.



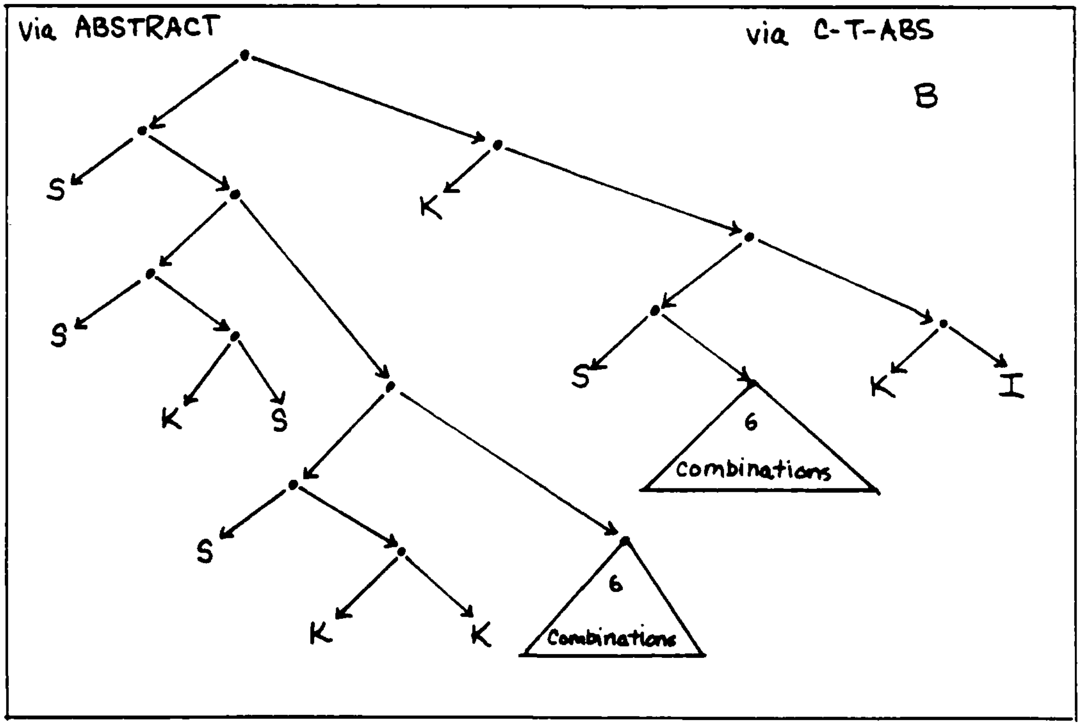
The ABSTRACTed and C-T-ABSed Versions of the λ-wff: λ x (+ x x)
Figure 2.14



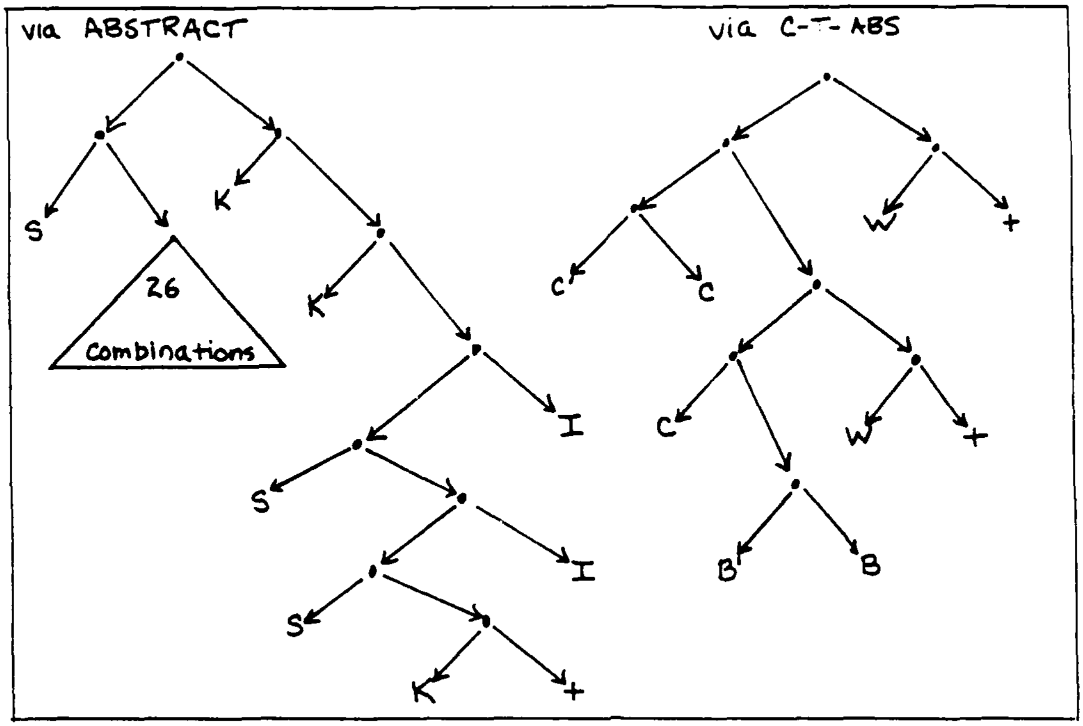
The ABSTRACTed and C-T-ABSed Versions of the λ-wff: λ f (λ x (f (f (f x))))
Figure 2.15



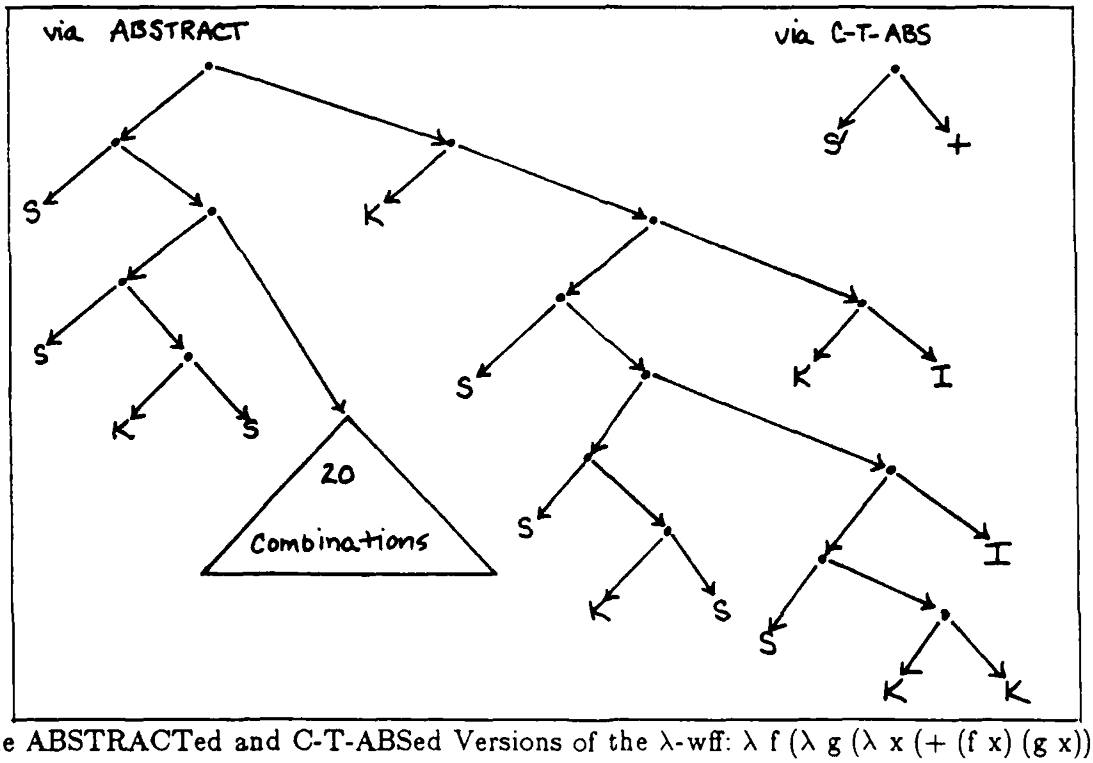
The ABSTRACTed and C-T-ABSed Versions of the λ-wff: λ x (λ f (f (f (f x))))
Figure 2.16



The ABSTRACTed and C-T-ABSed Versions of the λ-wff: λ f (λ g (λ x (f (g x))))
Figure 2.17

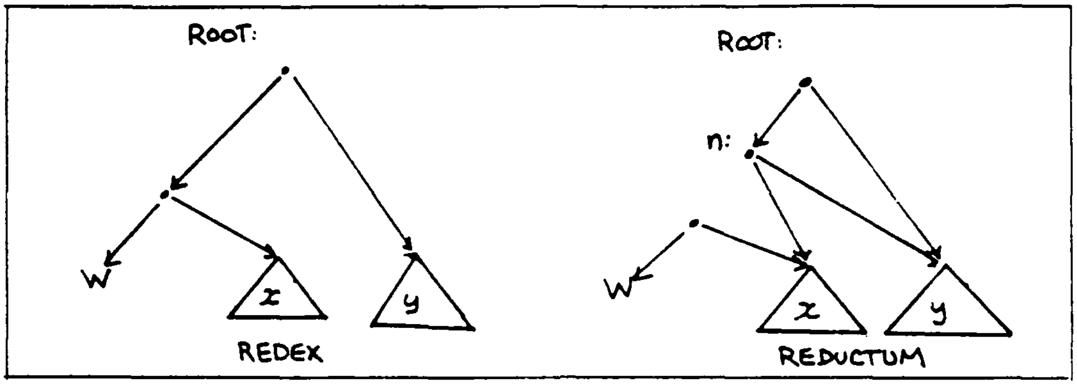


The ABSTRACTed and C-T-ABSed Versions of the λ-wff: λ f (λ x (λ y (f (+ x x) (+ y y))))
Figure 2.18

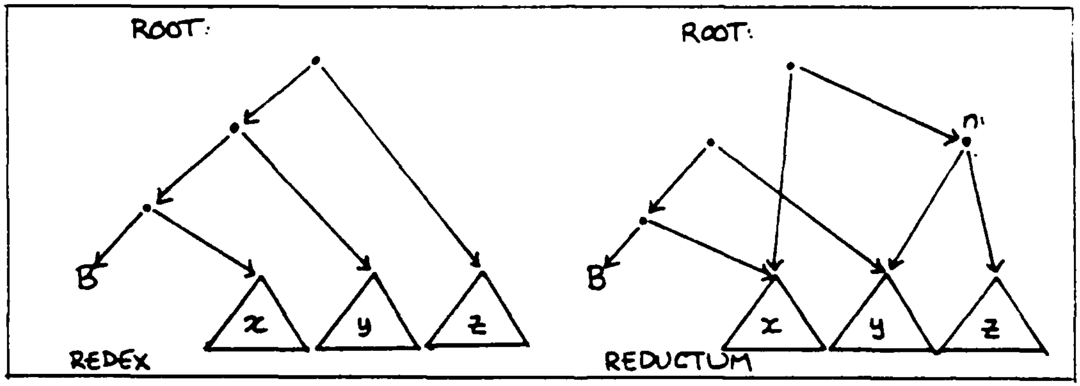


The ABSTRACTed and C-T-ABSed Versions of the λ-wff: λ f (λ g (λ x (+ (f x) (g x))))
Figure 2.19

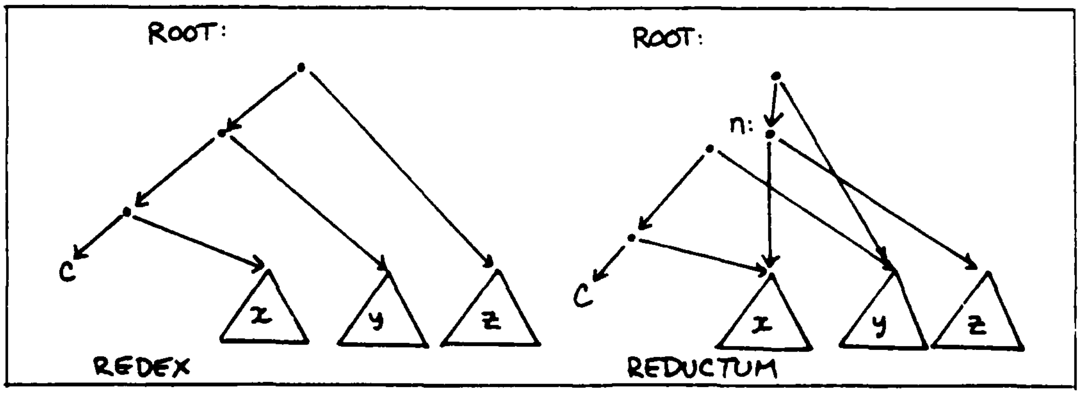
The graphical representations of the reduction rules for each of these new functors will now be displayed. The author believes that these pictures, although informal, may provide the reader with a better understanding of the workings of the rules than do the formal definitions. From the picture of the reduction rule for the functor **f**, one can infer the definitions of the predicate LNF-f-REDEX-P and the function LNF-f-REDUCTUM. As the author makes no use of the functors' formal definitions, these definitions will not be given.



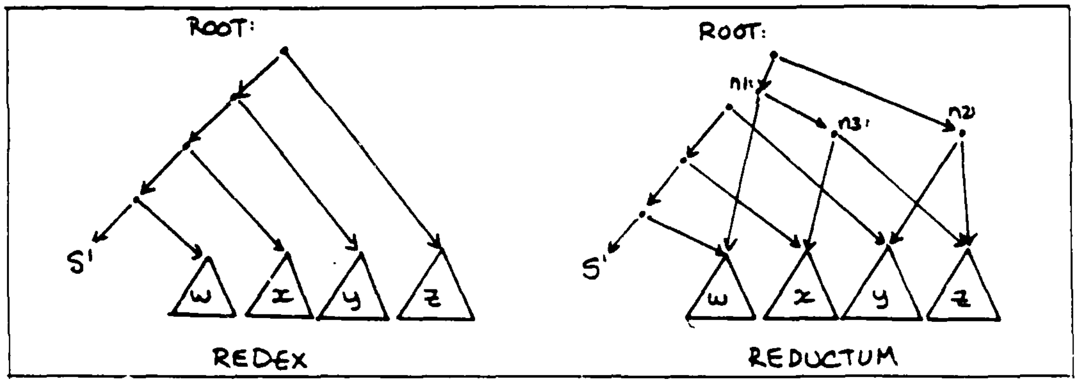
The Rule: W X Y → X Y Y
Figure 2.20



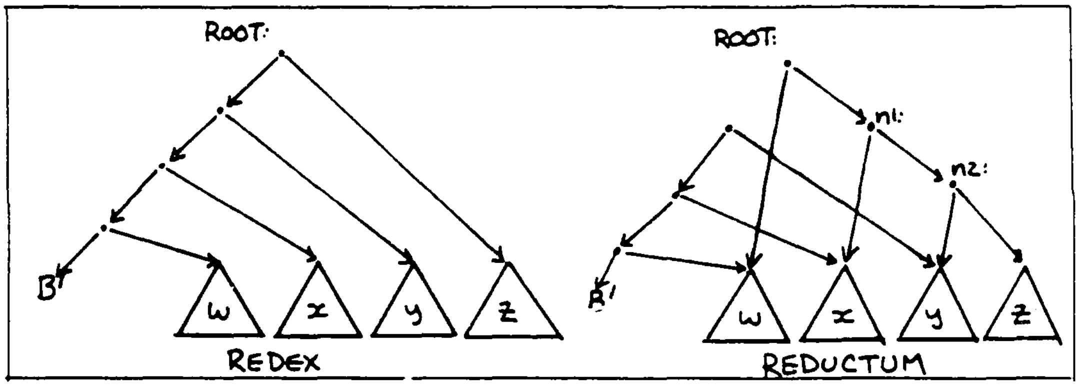
The Rule: B X Y Z → X (Y Z)
Figure 2.21



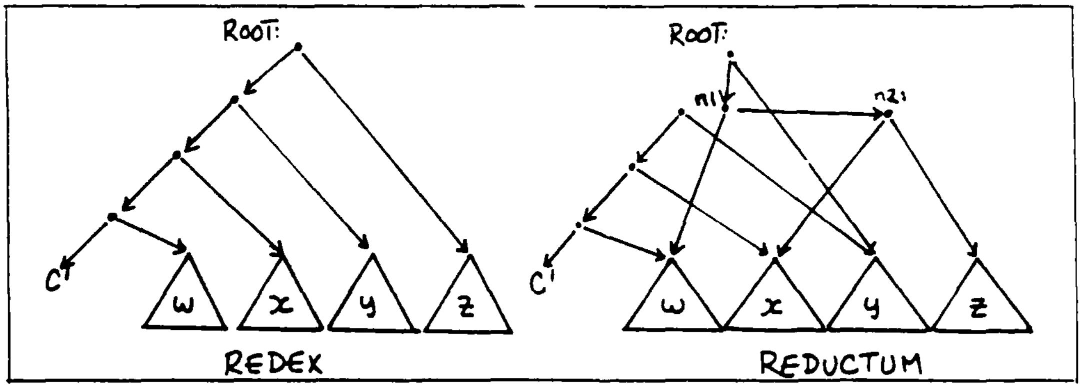
The Rule: C X Y Z → X Z Y
Figure 2.22



The Rule: S′ W X Y Z → W (X Z) (Y Z)
Figure 2.23



The Rule: B′ W X Y Z → W (X (Y Z))
Figure 2.24



The Rule: C′ W X Y Z → W (X Z) Y
Figure 2.25

2.2.3. Numeric Functors

Floating point numbers and integers are LNF constructors. Presented in this section are the LNF functors which manipulate the atomic LNF-wffs having constructors of this type as initial atoms.

The numeric functors are: NUMBERP, +, -, ×, DIV, IDIV, REM, EXP, <, >, ADD1, SUB1, and ZEROP. The formal reduction rules will be given only for ×, as the other functors' rules are almost identical. For these other functors, only the linearized reduction rules will be presented.

Definition 2.45: Let **X** be an LNF-wff. **X** is an *LNF-× redex* if *LNF-×-REDEX-P*[**X**] where:
LNF-×-REDEX-P[**X**] $\overset{\mathrm{Def}}{\Rightarrow}$
(*and* (*not* FORWARDED-P[ROOT[**X**],**X**])
INITIAL-ATOM[**X**] = ×
NUMBER-OF-ARGS[**X**] = 2
ARG[1,**X**] is an atom having a number as initial atom
ARG[2,**X**] is an atom having a number as initial atom)

Definition 2.46: Let **X** be an LNF-× redex. **Y** is the *LNF-× reductum of* **X** if *LNF-×-REDUCTUM*[**X**] = **Y** (**X** *LNF-×-imr* **Y**) where:
LNF-×-REDUCTUM[**X**] $\overset{\mathrm{Def}}{\Rightarrow}$
(*let* n_1 *be* INITIAL-ATOM[ARG[1,**X**]] *&*
n_2 *be* INITIAL-ATOM[ARG[2,**X**]] *in*
FORWARD-COMB[**X**,ATOMIC-GRAPH[$n_1 \times n_2$]])

The linear representation of ×'s substantive reduction rule is: × **n m** → $\underline{\mathbf{n} \times \mathbf{m}}$ [9]. This rule implies that the functor × has an arity of 2.

In addition to having a substantive reduction rule, × is also associated with the following two contextual reduction rules:

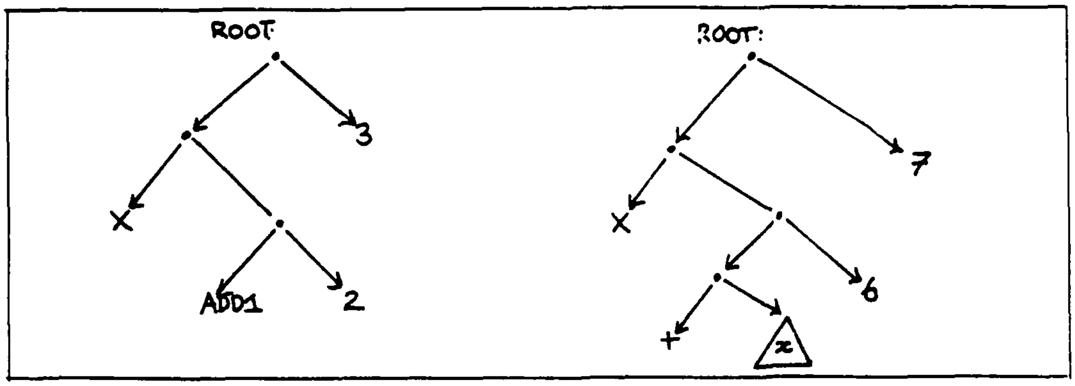
× **RDU X** → × **IMR X**
× **n RDU** → × **n IMR**

The first contextual rule expresses the relation: "in an LNF-wff having initial atom × and two arguments, the first of which is a reducible unknown, the unknown may be replaced with the wff to which it immediately reduces". The second rule states: "in an LNF-wff having initial atom × and two arguments, the first of which is a number and the second of which is a reducible unknown, the unknown may be replaced with the wff to which it immediately reduces".

Thus, both of these rules specify a context in which other reductions may take place. These contexts are called *functor specific reduction contexts* or, simply, *r-contexts*. Most of the new functors are associated with one or more contextual reduction rules specifying one or more r-contexts. The predicate, R-CONTEXT-P, takes three arguments: an LNF-wff **X**, a functor **f**, and a positive integer i. R-CONTEXT-P[**X**,**f**,i] is true iff "**X** is an **f** reduction context for argument i".

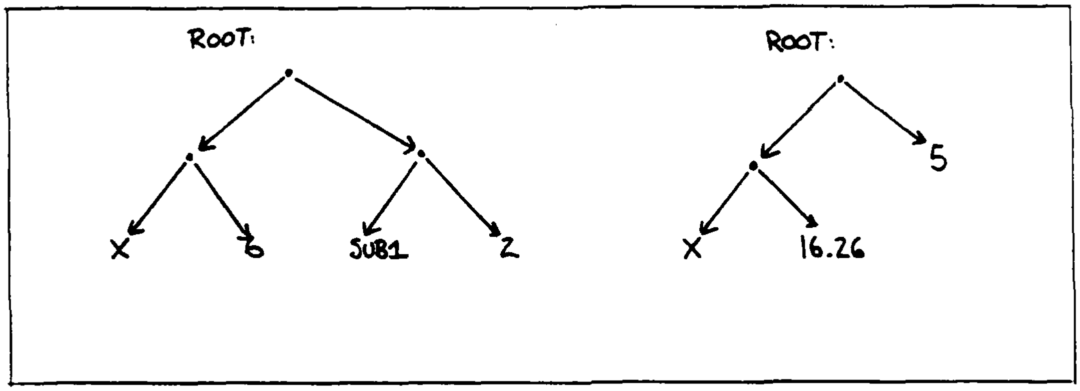
Some examples of × reduction contexts follow.

[9] When linearly displaying rules, expressions which are assumed to be evaluated by an agent outside the calculus appear underlined.



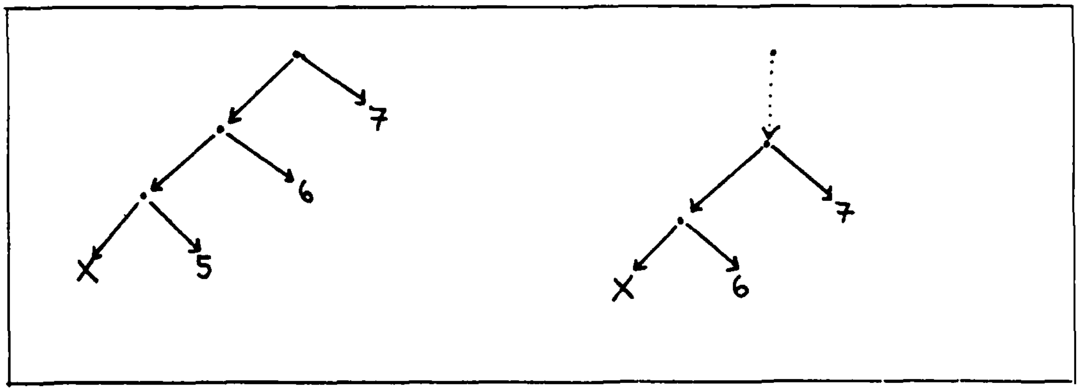
Two × Reduction Contexts for Argument 1
Figure 2.26

Note that a reduction context need not be reducible. A reduction context for argument i is reducible iff argument i is reducible.



Two × Reduction Contexts for Argument 2
Figure 2.27

Note also that an LNF-wff may be a redex as well as a reduction context. However, no LNF-wff which is a redex can be a reducible reduction context. If this were not the case then the LNF-calculus would be nondeterministic.



These are not×Reduction Contexts
Figure 2.28

A functor **f**, whose arguments must be reduced to lazy-normal form before its reduction rule may be applied, is a *strict* (sometimes called *totally strict*) functor. Some functors require that only some of their arguments be reduced before being applied. These functors are often referred to as *partially strict* or *strict in a specific argument(s)*. The functors × and NUMBERP are examples of strict functors. The functor IF, defined later, is strict in its first argument only.

The linearized reduction rules (both substantive and contextual) for all of the numeric functors are displayed below:[10]

NUMBERP	NUMBERP **n** → TRUE
	NUMBERP **CFN** → FALSE, if $\underline{\text{CFN not a number}}$
	NUMBERP **RDU** → NUMBERP **IMR**
+	+ **n m** → $\underline{\mathbf{n+m}}$
	+ **RDU Y** → + **IMR Y**
	+ **n RDU** → + **n IMR**
×	× **n m** → $\underline{\mathbf{n \times m}}$
	× **RDU Y** → × **IMR Y**
	× **n RDU** → × **n IMR**
-	- **n m** → $\underline{\mathbf{n-m}}$
	- **RDU Y** → - **IMR Y**
	- **n RDU** → - **n IMR**
DIV	DIV **n m** → $\underline{\mathbf{n/m}}$, if $\underline{\mathbf{m \neq 0}}$
	DIV **RDU Y** → DIV **IMR Y**
	DIV **n RDU** → DIV **n IMR**
IDIV	IDIV **i j** → $\underline{\text{integral quotient after } \mathbf{i/j}}$, if $\underline{\mathbf{j \neq 0}}$
	IDIV **RDU Y** → IDIV **IMR Y**
	IDIV **i RDU** → IDIV **i IMR**
REM	REM **n m** → $\underline{\text{remainder after } \mathbf{n/m}}$, if $\underline{\mathbf{m \neq 0}}$
	REM **RDU Y** → REM **IMR Y**
	REM **n RDU** → REM **n IMR**
EXP	EXP **i j** → $\underline{\text{the integer } \mathbf{i^j}}$, if $\underline{\mathbf{j \geq 0}}$
	EXP **i j** → $\underline{\text{the float } \mathbf{i^j}}$, if $\underline{\mathbf{j < 0}}$
	EXP **s i** → $\underline{\text{the float } \mathbf{s^i}}$
	EXP **n s** → $\underline{\text{the float } \mathbf{n^s}}$
	EXP **RDU Y** → EXP **IMR Y**
	EXP **n RDU** → EXP **n IMR**

[10] **Some rules take the form LHS → RHS**, if $\underline{\text{CONDITION}}$ where the CONDITION is an expression **to be evaluated by an outside agent**. Rules of this form should be read as saying "if CONDITION, **then LHS may be replaced by RHS**"

<	< **n m** → TRUE, if n<m < **n m** → FALSE, if n≥m < **RDU Y** → < **IMR Y** < **n RDU** → < **RDU IMR**
>	> **n m** → TRUE, if n>m > **n m** → FALSE, if n≤m > **RDU Y** → > **IMR Y** > **n RDU** → > **n IMR**
ADD1	ADD1 **n** → n+1 ADD1 **RDU** → ADD1 **IMR**
SUB1	SUB1 **n** → n-1 SUB1 **RDU** → SUB1 **IMR**
ZEROP	ZEROP **n** → n=0 ZEROP **RDU** → ZEROP **IMR**

Note that, in all cases, only one rule, be it substantive or contextual, would be applicable to any LNF-wff. Note also that, for each functor f, all of f's reduction rules require the same number of arguments. There are no LNF functors having multiple arities.

2.2.4. Boolean Functors

The boolean constructors are TRUE and FALSE. The boolean functors are: BOOLEANP, NOT, OR, and AND. Their linearized reduction rules are displayed below:

BOOLEANP	BOOLEANP **b** → TRUE BOOLEANP **CFN** → FALSE, if CFN not a boolean BOOLEANP **RDU** → BOOLEANP **IMR**
NOT	NOT TRUE → FALSE NOT FALSE → TRUE NOT **RDU** → NOT **IMR**
OR	OR TRUE **Y** → TRUE OR FALSE **b** → **b** OR FALSE **RDU** → OR FALSE **IMR** OR **RDU Y** → OR **IMR Y**
AND	AND FALSE **Y** → FALSE AND TRUE **b** → **b** AND TRUE **RDU** → AND TRUE **IMR** AND **RDU Y** → AND **IMR Y**

The formal definition of OR's substantive reduction rules will now be presented.

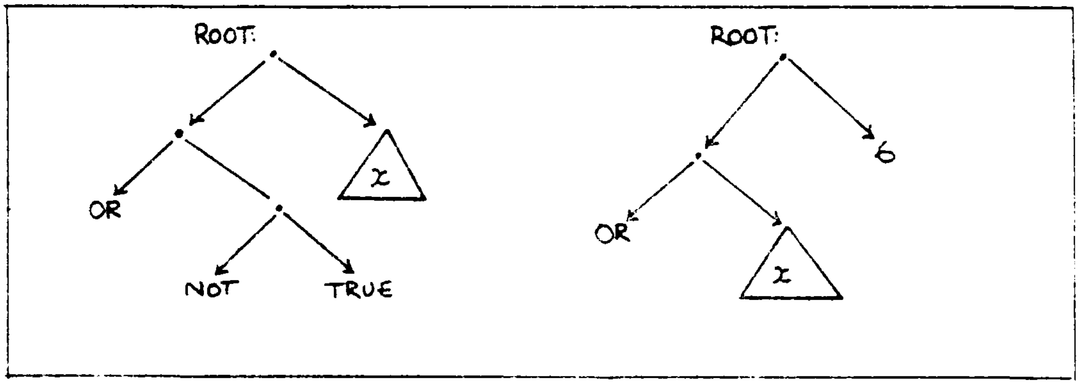
Definition 2.47: Let **X** be an LNF-wff. **X** is an *LNF-OR redex* if *LNF-OR-REDEX-P*[**X**] where:

```
LNF-OR-REDEX-P[X] ≝
(and (not FORWARDED-P[ROOT[X],X])
    INITIAL-ATOM[X] = OR
    NUMBER-OF-ARGS[X] = 2
    ATOM-P[ARG[1,X]]
    (or TRUE = INITIAL-ATOM[ARG[1,X]]
        (and FALSE = INITIAL-ATOM[ARG[1,X]]
            ARG[2,X] is an atom having a truthvalue
                as initial atom)))
```

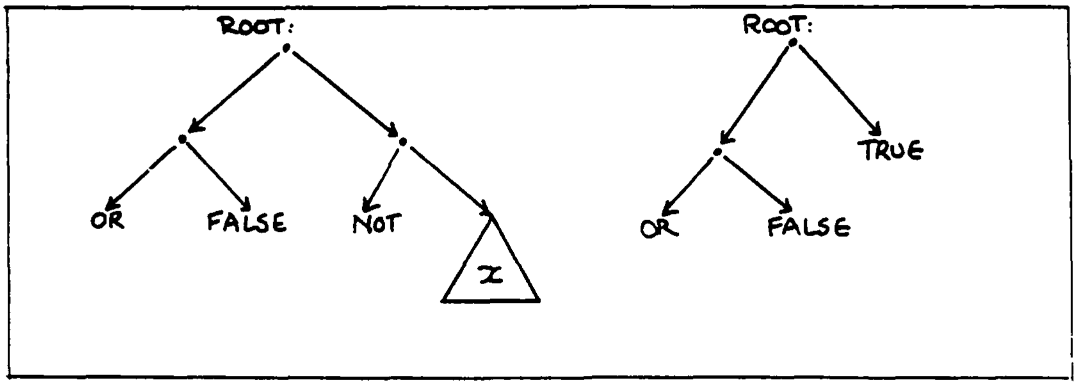
Definition 2.48: Let **X** be an LNF-OR redex. **Y** is the *LNF-OR reductum of* X if *LNF-OR-REDUCTUM*[**X**] = **Y** (**X** *LNF-OR-imr* Y) where:

```
LNF-OR-REDUCTUM[X] ≝
 (if INITIAL-ATOM[ARG[1,X]] = TRUE
  then FORWARD-COMB[X,ARG[1,X]]
  else ;; INITIAL-ATOM[ARG[1,X]] = FALSE and
      ;; INITIAL-ATOM[ARG[2,X]] a truthvalue
   FORWARD-COMB[X,ARG[2,X]])
```

The functor OR is also associated with two contextual reduction rules. Some examples of OR reduction contexts follow.



Two OR Reduction Contexts for Argument 1
Figure 2.29



Two OR Reduction Contexts for Argument 2
Figure 2.30

The realizations of the functors OR and AND (presented in the next chapter) perform like ML's OrElse and AndThen boolean operators. LISP implementations also provide boolean connectives whose arguments are evaluated as required.

2.2.5. Pair and List Oriented Functors

Lists are data structures familiar to all functional programmers. Since lists are so commonly used, some functors have been defined which manipulate them. There are two constructors which are used to make lists: [ ] and PAIR. The constructor [ ] is used to make empty (or null) lists and PAIR is used to build pairs. A (linearized) list is either:

the null list [ ], or
the pair (PAIR **X** L), where L is also a list.

The LNF-wff **X** is called the head of the list (PAIR **X** L). L is called its tail. The PAIR constructor may, of course, also be used to pair other types of LNF-wffs.

The pair and list oriented functors are: HD, TL, NULLP, PAIRP, NTH, APPEND, MAP, MEMBER, COLLECT, FILTER, REM-DUPS, REM-DUPS′, FB, FB′, FBT, FBT′, INTERLEAVE, FLATMAP, ENUMERATE, UP, DOWN, and TURN. The utility of some of the functors presented in this section is apparent. For many others, however, the reasons for including them into the calculus are not so obvious. The uses to which these functors are put, which justify their inclusion in the calculus, are presented in the next chapter. Their linearized reduction rules are displayed below:

HD HD (PAIR **X Y**) → **X**
HD **RDU** → HD **IMR**

TL TL (PAIR **X Y**) → **Y**
TL **RDU** → TL **IMR**

NULLP NULLP [ ] → TRUE
NULLP **CFN** → FALSE, if CFN≠[ ]
NULLP **RDU** → NULLP **IMR**

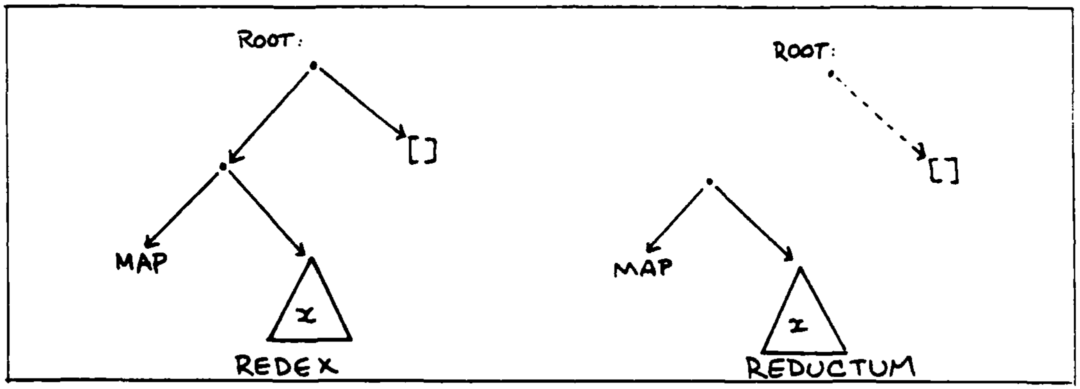
PAIRP	PAIRP (PAIR **X Y**) → TRUE PAIRP **CFN** → FALSE, if CFN not a pair PAIRP **RDU** → PAIRP **IMR**
NTH	NTH 1 (PAIR **X Y**) → **X** NTH **i** (PAIR **X Y**) → NTH i-1 **Y**, if i≥1 NTH **RDU Y** → NTH **IMR Y** NTH **i RDU** → NTH **i IMR**, if i≥0
APPEND	APPEND [ ] [ ] → [ ] APPEND [ ] **P** → **P** APPEND (PAIR **X Y**) **Z** → PAIR **X** (APPEND **Y Z**) APPEND **RDU Y** → APPEND **IMR Y**
INTERLEAVE	INTERLEAVE [ ] **P** → **P** INTERLEAVE **P** [ ] → **P** INTERLEAVE (PAIR **X Y**) **P** → PAIR **X** (INTERLEAVE **P Y**) INTERLEAVE **RDU Y** → INTERLEAVE **IMR Y** INTERLEAVE **P RDU** → INTERLEAVE **P IMR**
FLATMAP	FLATMAP **X** [ ] → [ ] FLATMAP **X** (PAIR **Y Z**) → INTERLEAVE (**X Y**) (FLATMAP **X Z**) FLATMAP **X RDU** → FLATMAP **X IMR**
ENUMERATE	ENUMERATE **X** → TURN [ ] **X**
TURN	TURN **X** [ ] → UP **X** [ ] [ ] TURN **X** (PAIR **Y Z**) → UP (PAIR **Y X**) [ ] **Z** TURN **X RDU** → TURN **X IMR**
UP	UP [ ] **X Y** → DOWN **X** [ ] **Y** UP (PAIR [ ] **X**) **Y Z** → UP **X Y Z** UP (PAIR (PAIR X_1 X_2) **Y**) **W Z** → PAIR X_1 (UP **Y** (PAIR X_2 **W**) **Z**) UP (PAIR **RDU X**) **Y Z** → UP (PAIR **IMR X**) **Y Z** UP **RDU Y Z** → UP **IMR Y Z**

DOWN	DOWN [ ] [ ] [ ] → [ ] DOWN [ ] **P** [ ] → UP **P** [ ] [ ] DOWN [ ] **RDU** [ ] → DOWN [ ] **IMR** [ ] DOWN [ ] **X** (PAIR [ ] **Y**) → TURN **X Y** DOWN [ ] **X** (PAIR (PAIR $\mathbf{Y}_1$ $\mathbf{Y}_2$) **Z**) → PAIR $\mathbf{Y}_1$ (TURN (PAIR $\mathbf{Y}_2$ **X**) **Z**) DOWN [ ] **Y RDU** → DOWN [ ] **Y IMR** DOWN [ ] **Y** (PAIR **RDU W**) → DOWN [ ] **Y** (PAIR **IMR W**) DOWN (PAIR [ ] **X**) **Y Z** → DOWN **X Y Z** DOWN (PAIR (PAIR $\mathbf{X}_1$ $\mathbf{X}_2$) **Y**) **Z W** → PAIR $\mathbf{X}_1$ (DOWN **Y** (PAIR $\mathbf{X}_2$ **Z**) **W**) DOWN (PAIR **RDU X**) **Y Z** → DOWN (PAIR **IMR X**) **Y Z** DOWN **RDU Y Z** → DOWN **IMR Y Z**
MAP	MAP **X** [ ] → [ ] MAP **X** (PAIR **Y Z**) → PAIR (**X Y**) (MAP **X Z**) MAP **X RDU** → MAP **X IMR**
MEMBER	MEMBER [ ] **X** → FALSE MEMBER (PAIR **X Y**) **Z** → IF (= **X Z**) TRUE (MEMBER **Y Z**) MEMBER **RDU Y** → MEMBER **IMR Y**
COLLECT	COLLECT [ ] **X Y** → **Y** COLLECT (PAIR **X Y**) **W Z** → **W X** (COLLECT **Y W Z**) COLLECT **RDU Y Z** → COLLECT **IMR Y Z**
FILTER	FILTER **X** [ ] → [ ] FILTER **X** (PAIR **Y Z**) → IF (**X Y**) (PAIR **Y** (FILTER **X Z**)) (FILTER **X Z**) FILTER **X RDU** → FILTER **X IMR**
REM-DUPS	REM-DUPS **X** → REM-DUPS′ **X** [ ]
REM-DUPS′	REM-DUPS′ [ ] **X** → **X** REM-DUPS′ (PAIR **X Y**) **Z** → IF (MEMBER **Z X**) (REM-DUPS′ **Y Z**) (PAIR **X** (REM-DUPS′ **Y Z**)) REM-DUPS′ **RDU Y** → REM-DUPS′ **IMR Y**
FB	FB **n m** → PAIR **n** (FB′ **n+m** **m**), if **m≠0** FB **n m** → PAIR **n** (PAIR **n** ...), if **m=0** FB **RDU Y** → FB **IMR Y** FB **n RDU** → FB **n IMR**
FB′	FB′ **n m** → PAIR **n** (FB′ **n+m** **m**)

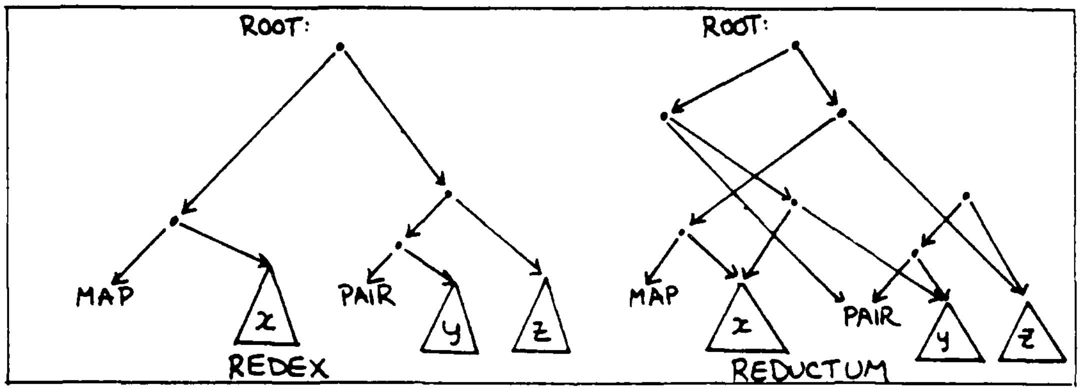
FBT FBT **n m o** → PAIR **n** (FBT′ $\underline{\mathbf{n+m}}$ **m o**),
if $\underline{(\mathbf{m}\geq 0 \text{ and } \mathbf{n}\leq \mathbf{o}) \text{ or } (\mathbf{m}<0 \text{ and } \mathbf{n}\geq \mathbf{o})}$
FBT **n m o** → [ ],
if $\underline{(\mathbf{m}>\mathbf{0} \text{ and } \mathbf{n}>\mathbf{o}) \text{ or } (\mathbf{m}<\mathbf{0} \text{ and } \mathbf{n}<\mathbf{o})}$
FBT **n m o** → PAIR **n** (PAIR **n** ...), if $\underline{\mathbf{m}=0}$
FBT **RDU Y Z** → FBT **IMR Y Z**
FBT **n RDU Z** → FBT **n IMR Z**
FBT **n m RDU** → FBT **n m IMR**

FBT′ FBT′ **n m o** → PAIR **n** (FBT′ $\underline{\mathbf{n+m}}$ **m o**),
if $\underline{(\mathbf{m}>0 \text{ and } \mathbf{n}\leq \mathbf{o}) \text{ or } (\mathbf{m}<0 \text{ and } \mathbf{n}\geq \mathbf{o})}$
FBT′ **n m o** → [ ],
if $\underline{(\mathbf{m}>\mathbf{0} \text{ and } \mathbf{n}>\mathbf{o}) \text{ or } (\mathbf{m}<\mathbf{0} \text{ and } [\mathbf{n}<\mathbf{o})}$

Although no formal definitions will be given for these functors, pictures of MAP's two substantive reduction rules will be displayed.

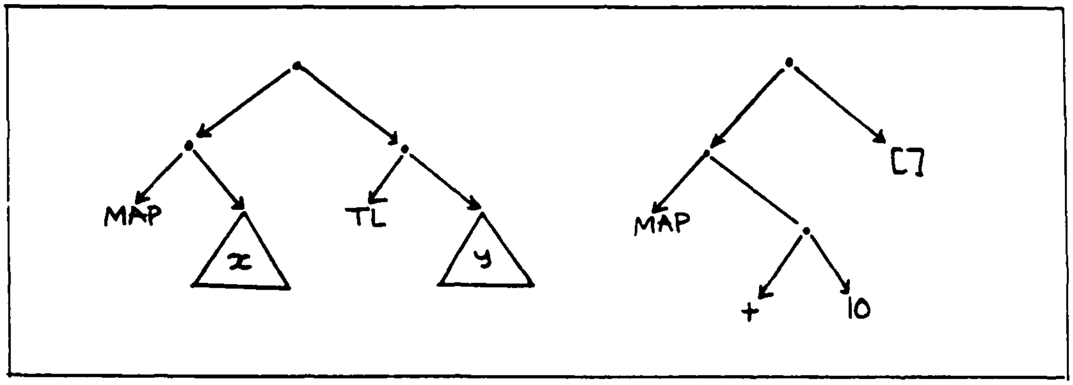


The Rule: MAP **X** [ ] → [ ]
Figure 2.31



The Rule: MAP **X** (PAIR **Y Z**) → PAIR (**X Y**) (MAP **X Z**)
Figure 2.32

Note that the combination (MAP **X**) in the redex is "reused" in the reductum. The following figure contains two examples of MAP reduction contexts for argument 2 as specified by MAP's contextual reduction rule. Note that there is no MAP reduction context for argument 1.



Two Examples of MAP Reduction Contexts for Argument 2
Figure 2.33

2.2.6. Miscellaneous Functors

The remaining LNF functors are presented in this section. They are: Y, =, L, IF, UNKNOWNP, CONSTRUCTIONP, FUNCTIONP, FUNCTOR, ARITY, CONSTRUCTOR, NUM-ARGS, ARG, ATOMP, COMBINATIONP, OPERATOR, OPERAND, A-S-E, A-S-E′, A-S, A-S′, and APP-TO-ARGS. Presented below are their associated reduction rules.

It is not expected that the reader immediately appreciates the usefulness of the functors: A-S-E, A-S-E′, A-S, A-S′, and APP-TO-ARGS. Their existence in the calculus is justified in Chapter 3.

Y $\quad \mathbf{Y\,X \rightarrow X\,(X\,(X\,...))}$

= $\quad \mathbf{= cf_1\ cf_2 \rightarrow \underline{cf_1{=}cf_2}}$

$\mathbf{= CFN_1\ CFN_2 \rightarrow}$
$\mathbf{AND\ (=\ (OPERATOR\ CFN_1)\ (OPERATOR\ CFN_2))}$
$\mathbf{(=\ (OPERAND\ CFN_1)\ (OPERAND\ CFN_2))}$

$\mathbf{= RDU\ Y \rightarrow\ = IMR\ Y}$

$\mathbf{= CFN\ RDU \rightarrow CFN\ IMR}$

Note that ='s reduction rules permit comparison of functions as well as constructions. Two functions (constructions) are equal, the rules specify, if and only if they have the same normal form. Thus, the functor = (when applied to functions) is testing for *definitional equality* and not *extensional equality* — i.e. it's testing to see if two functions are the same algorithm.

L — L **cf CFN** → TRUE, if NUM-ARGS[CFN]>0
L **CFN cf** → FALSE, if NUM-ARGS[CFN]>0
L $\mathbf{cf}_1$ $\mathbf{cf}_2$ →
$\mathbf{cf}_1$ lexicographically less than $\mathbf{cf}_2$
L $\mathbf{CFN}_1$ $\mathbf{CFN}_2$ →
OR (L (OPERATOR $\mathbf{CFN}_1$) (OPERATOR $\mathbf{CFN}_2$))
(AND (= (OPERATOR $\mathbf{CFN}_1$) (OPERATOR $\mathbf{CFN}_2$))
(L (OPERAND $\mathbf{CFN}_1$) (OPERAND $\mathbf{CFN}_2$))),
if $\mathbf{CFN}_1$ and $\mathbf{CFN}_2$ are both combinations
L **RDU Y** → L **IMR Y**
L **CFN RDU** → L **CFN IMR**

The functor L imposes a total ordering on the set: Functions ∪ Constructions

IF — IF **TRUE X Y** → **X**
IF **FALSE X Y** → **Y**
IF **RDU X Y** → IF **IMR X Y**

UNKNOWNP — UNKNOWNP **CFN** → FALSE
UNKNOWNP **IRU** → TRUE
UNKNOWNP **RDU** → UNKNOWNP **IMR**

FUNCTIONP — FUNCTIONP **FN** → TRUE
FUNCTIONP **CN** → FALSE
FUNCTIONP **RDU** → FUNCTIONP **IMR**

FUNCTOR — FUNCTOR **FN** → INITIAL-ATOM[FN]
FUNCTOR **RDU** → FUNCTOR **IMR**

CONSTRUCTIONP — CONSTRUCTIONP **CN** → TRUE
CONSTRUCTIONP **FN** → FALSE
CONSTRUCTIONP **RDU** → CONSTRUCTIONP **IMR**

CONSTRUCTOR — CONSTRUCTOR ($\mathbf{c}\ \mathbf{X}_1 \cdots \mathbf{X}_n$) → **c**
CONSTRUCTOR **RDU** → CONSTRUCTOR **IMR**

ARITY — ARITY **FN** →
ARITY[INITIAL-ATOM[FN]] - NUM-ARGS[FN]
ARITY **RDU** → ARITY **IMR**

NUM-ARGS — NUM-ARGS **CFN** → NUM-ARGS[CFN]
NUM-ARGS **RDU** → NUM-ARGS **IMR**

ARG — ARG **i CFN** → ARG[i,CFN]
, if 1≤i≤NUM-ARGS[CFN]
ARG **RDU Y** → ARG **IMR Y**
ARG **i RDU** → ARG **i IMR**

ATOMP	ATOMP CFN → NUM-ARGS[CFN]=0 ATOMP RDU → ATOMP IMR
COMBINATIONP	COMBINATIONP CFN → NUM-ARGS[CFN]>0 COMBINATIONP RDU → COMBINATIONP IMR
OPERATOR	OPERATOR CFN → OPERATOR[CFN] OPERATOR RDU → OPERATOR IMR
OPERAND	OPERAND CFN → OPERAND[CFN] OPERAND RDU → OPERAND IMR
A-S-E	A-S-E c i X Y (c $Z_1 \cdots Z_i$) → X A-S-E c_1 i X Y (c_2 $Z_1 \cdots Z_j$) → Y, if $c_1 \neq c_2$ or $i \neq j$ A-S-E c i X Y FN → Y A-S-E c i X Y RDU → A-S-E c i X Y IMR
A-S-E′	A-S-E′ c i X Y (c $Z_1 \cdots Z_i$) → X A-S-E′ c_1 i X Y (c_2 $Z_1 \cdots Z_j$) → Y, if $c_1 \neq c_2$ or $i \neq j$ A-S-E′ c i X Y FN → Y
A-S	A-S c i X (c $Z_1 \cdots Z_i$) → X $Z_1 \cdots Z_i$ A-S c i X RDU → A-S c i X IMR
A-S′	A-S′ c i X (c $Z_1 \cdots Z_i$) → X $Z_1 \cdots Z_i$
APP-TO-ARGS	APP-TO-ARGS i X Y → X (ARG 1 Y) ... (ARG i Y)

Most of the miscellaneous functors have formal definitions similar to those whose definitions have already been presented. The functor Y, the "fixed-point finding functor", however, has a definition that is a little different and therefore will be displayed. Y is called the fixed-point finding functor since its characteristic property is that:

for all functions F, Y F is a fixed-point of F, i.e.
Y F = F (Y F).[11]

By repeatedly substituting the wff F (Y F) for occurrences of the wff Y F on the right hand side of the equal sign, one gets the equation:

$$Y\ F = F\ (F\ (F\ \ldots))$$

which looks like the linearized rule for Y. This linearized rule is one which is deceptive. A cycle exists in the reductum which cannot be displayed in this linear format. The formal definition and a graphical picture of Y's reduction rule follow.

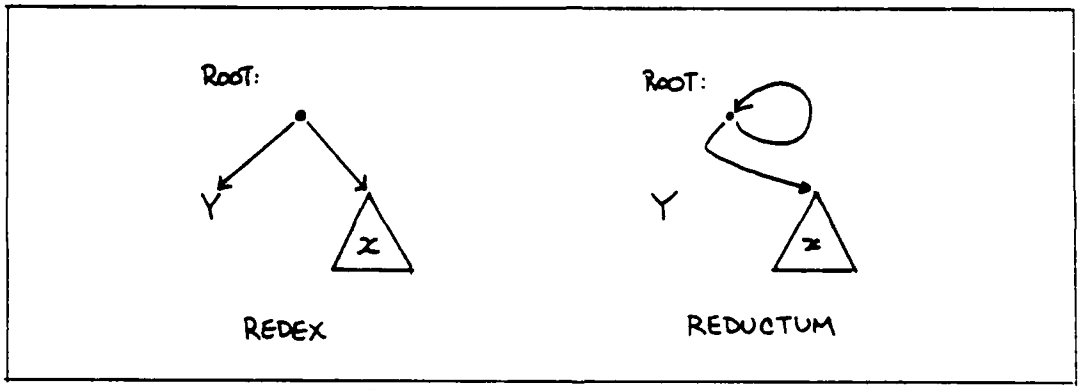
[11] The functor Y plays an important part in the implementation This role will be discussed in Chapter 3.

Definition 2.49: Let **X** be an LNF-wff. **X** is an *LNF-Y redex* if *LNF-Y-REDEX-P*[**X**] where:

```
LNF-Y-REDEX-P[X] $\overset{\text{Def}}{\Rightarrow}$
(and (not FORWARDED-P[ROOT[X],X])
     INITIAL-ATOM[X] = Y
     NUMBER-OF-ARGS[X] = 1)
```

Definition 2.50: Let **X** be an LNF-Y redex. **Y** is the *LNF-Y reductum of* **X** if *LNF-Y-REDUCTUM*[**X**] = **Y** (**X** *LNF-Y-imr* **Y**) where:

```
LNF-Y-REDUCTUM[X] $\overset{\text{Def}}{\Rightarrow}$
 (let root be ROOT[X] in
   <VS[X],
     RATOR[X]|(VS[X]-{root}) ∪ {<root,RAND[root]>},
     RAND[X]|(VS[X]-{root}) ∪ {<root,root>},
     FWD[X],
     ATOM[X],
     root>)
```



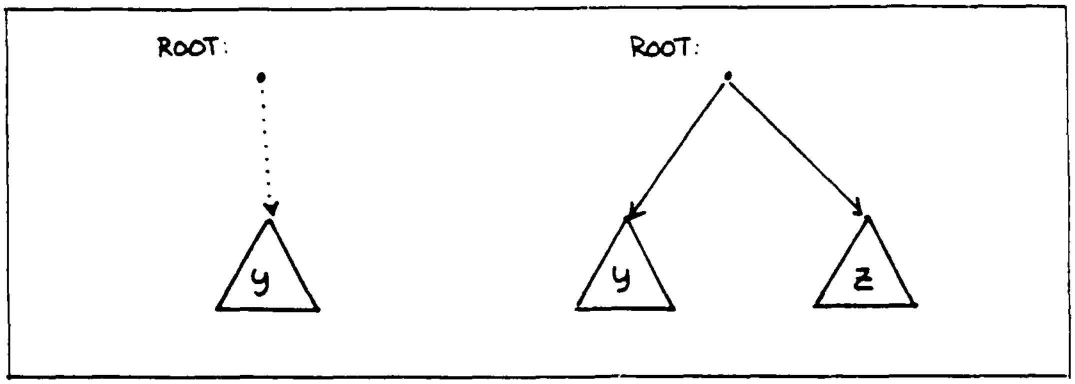
An Example of LNF Y Reduction
Figure 2.34

The LNF-calculus' functors have been presented. In Appendix A, all of the LNF-calculus' linearized reduction rules are redisplayed. They are displayed in two groups — first the substantive reduction rules, then the contextual reduction rules.

2.2.7. Reduction

Informally, an LNF-wff **X** is reducible (there is another LNF-wff to which it immediately reduces) if either **X** is a redex or **X** is a reducible reduction context (i.e. **X** is a context which permits reduction of one of its subformulas (**Y**), and **Y** is reducible).

All redex-reductum pairs are specified by the calculus' substantive reduction rules. The functor specific reduction contexts are specified by the calculus' contextual reduction rules. In addition to these r-contexts, two other reduction contexts (which are not functor specific) exist. An LNF-wff **X** which is forwarded to the LNF-wff **Y** is a reduction context for **Y**. A combination **X** having operator **Y** is also a reduction context for **Y**. These two reduction contexts are graphically displayed below.



Two Reduction Contexts for the LNF-wff Labeled **Y**
Figure 2.35

Definition 2.51: Let **X** be an LNF-wff and let v be in VS[**X**]. The *LNF-wff described in* **X** *rooted at* v is (*LNF-WFF*[**X**,v]) where

```
LNF-WFF[X,v] $\overset{\mathrm{Def}}{\Rightarrow}$
  <VS[X],RATOR[X],RAND[X],FWD[X],ATOM[X],v>
```

The formal definition of the LNF-calculus' "immediately reducible to" relation follows.

Definition 2.52: Let **X** and **Y** be LNF-wffs. **X** *immediately reduces to* **Y** iff **X** *LNF-imr* **Y** where

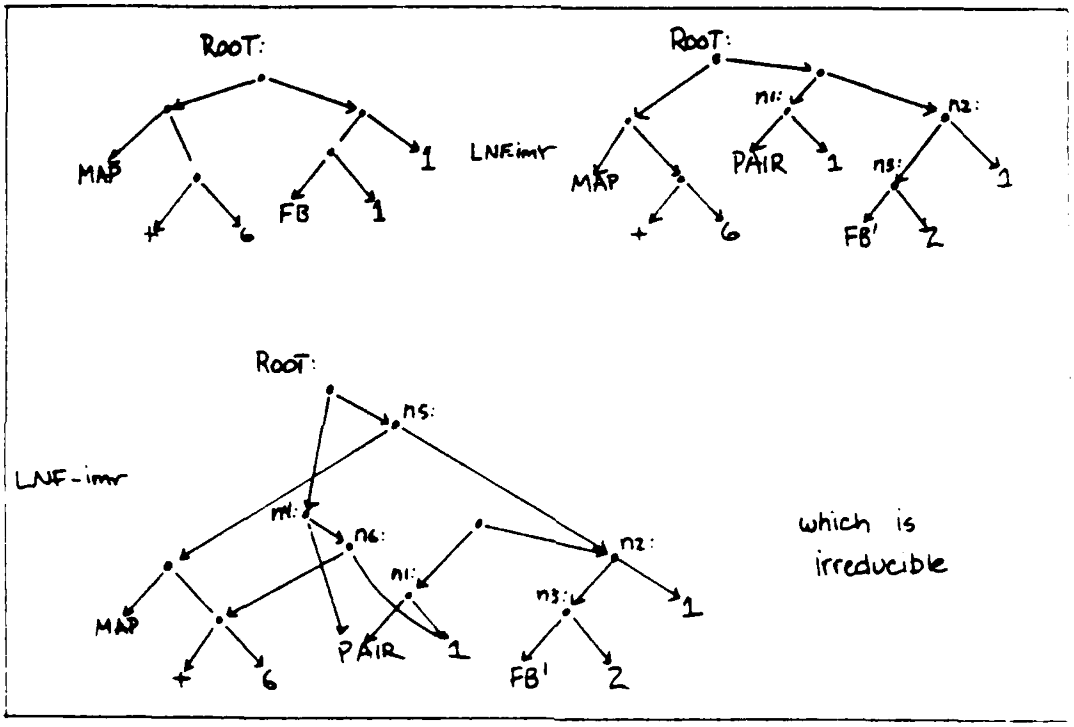
```
X LNF-imr Y $\overset{\mathrm{Def}}{\Rightarrow}$
  (let xroot be ROOT[X] in
    (if FORWARDED-P[xroot,X]
      then (let yroot be ROOT[Y] in
             (and FORWARDED-P[yroot,Y]
                  xroot = yroot
                  (SUBFORMULA[X,FORWARDED-TO[xroot,X]]
                    LNF-imr
                    SUBFORMULA[Y,FORWARDED-TO[yroot,Y]])))
      else
        (or (there is an LNF functor f s.t.
              (and LNF-f-REDEX-P[X]
                   Y = LNF-f-REDUCTUM[X]))
            (there is an LNF functor f and an i s.t.
              (and $1 \leq i \leq$ NUM-ARGS[X]
                   R-CONTEXT-P[X,f,i]
                   ARG[i,X] is reducible
                   Y = REDUCED-R-CONTEXT[X.i]))
            (there is an LNF-wff Z s.t
              (and OPERATOR[X] LNF-imr Z
                   Y = LNF-WFF[Z,xroot])))))
```

Definition 2.53: Let **X** be a reducible reduction context for argument **i** — i.e. there is some functor **f** such that R-CONTEXT-P[X,f,i] and ARG[i,X] is reducible. Performing one LNF reduction on **X** yields the LNF-wff: *REDUCED-R-CONTEXT*[X,i], where:

REDUCED-R-CONTEXT[X,i] $\overset{\text{Def}}{=}$
(*let* RARG *be* the LNF-wff such that
ARG[i,X] LNF-imr RARG *in* LNF-WFF[RARG,ROOT[X]])

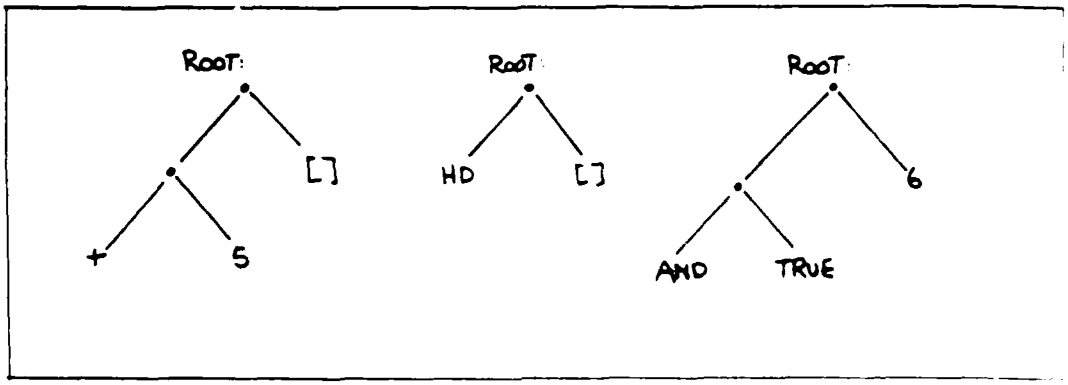
Definition 2.54: *LNF-red* is the transitive closure of LNF-imr.

Definition 2.55: *LNF-red** is the reflexive transitive closure of LNF-imr.



An Example of an LNF Reduction Sequence
Figure 2.36

Note that an irreducible LNF-wff may contain redexes. Irreducible LNF-wffs are said to be in *lazy-normal form*. It may be noted that all constructions and all functions are in lazy-normal form. Constructions are in lazy-normal form since all reduction rules (both substantive and contextual) require a functor as initial atom. Functions are in lazy-normal form because, although they have a functor as initial atom, they do not have enough arguments to form either a redex or a reduction context. Thus, all reducible LNF-wffs must be unknowns. Not all unknowns are reducible, however. Some irreducible unknowns are displayed below.



Some Irreducible Unknowns
Figure 2.37

2.3. Summary

Two deterministic (and therefore trivially Church-Rosser) graph oriented reduction calculi, the SKI-G-calculus and the LNF-calculus have been presented. The SKI-G-calculus is a formalized version of D.A. Turner's "normal order graph reduction" machine. Its definition is similar in form to C.P. Wadsworth's definition of the λ-G-calculus. Although equivalent in power to the λ-calculus it has been argued that the SKI-G-calculus would not do as a model for an implementation of an FP language. The LNF-calculus was presented as a calculus which would enjoy all of the advantages of the SKI-G-calculus (no variables, structure sharing). In the following chapter the realization of the LNF-calculus in ZetaLisp on a Lisp machine is discussed in detail.

Chapter 3

An Experimental Implementation of the LNF Language

LNF is a fully lazy, higher order, purely functional language with reduction semantics.

A language which is fully lazy is one which [illegible] evaluates expressions [illegible] called normal or leftmost order and never evaluates an expression more than once. A language which performs normal order reduction evaluation but does not ensure that each expression is reduced at most once is merely lazy. In a language which is fully lazy infinite objects may be dealt with naturally. For more discussion of the benefits of a fully lazy language the reader is referred to [Wadsworth 1971] (Wadsworth called this reduction order call by need), [Henderson 1978], [Friedman 1976] and [Hughes 1982]. A language which is higher order is one which treats functions as objects as first class citizens (they may be passed as arguments to other functions and returned as results just like any other object in the language). See [Turner 1979] and [Turner 1982] for more on the advantages of higher order programming. A language which is purely functional (also referred to as purely applicative) is one which is referentially transparent [illegible]. A language which enjoys the property of referential transparency is different [illegible] expression [illegible] the same [illegible] with assignment [illegible] FORTRAN and sadly most LISPs [illegible] referentially transparent. There are many reasons why it is advantageous to use a referentially transparent language but the most important may be that expressions in that language may be reduced in any order, even simultaneously. See [Backus 1978], [Turner 1981a], [Turner 1981b] and [Turner 1982] for more arguments in favor of referentially transparent languages. A language *with reduction semantics* is one which accepts expressions as input and produces (irreducible) expressions as output. The definition of irreducibility varies from language to language. In the LNF language an expression is irreducible if and only if it is in lazy-normal form.

The LNF programming environment was developed to give the author "hands on" experience with the issues involved in implementing such a language. What follows is a description of the implementation.

3.1. System Organization

The user interface to the system is a listen-respond loop not unlike the user interfaces present in most Lisp implementations. The user provides the system with two kinds of input: expressions and directives.

Presented with a well-formed LNF expression E (which is different from an LNF-wff), the system performs the following:

(Display (LNF-of-wff (Compile E)))

Compile takes well-formed LNF expressions as input and produces LNF-wffs as output. LNF-of-wff accepts LNF-wffs as input and produces LNF-wffs in lazy-normal form as output.[1] Display (sometimes working with LNF-of-wff) outputs to the terminal the reduced[2] LNF-wff in a linearized format. Each of these operations (Compile, LNF-of-wff, and Display) is described in detail in this chapter.

Directives given to the system modify the system. For example, there are directives which change how the system displays returned expressions, enable reduction monitoring and tracing, allow the user to give names to LNF expressions, start/end the recording of a session on a file, etc. Directives are input via a mouse device while expressions are typed at a system prompt.[3]

3.2 ZetaLisp Representation of LNF-wffs

LNF-wffs are represented in a straightforward way using ZetaLisp symbols, conses, and arrays.

An Atomic LNF-wff -- i.e. a constructor or a functor -- is represented in the machine by the ZetaLisp symbol having the same name. On the property list of the symbol representing each functor, both the functor's arity and a routine which is an encoding of the functor's reduction rule(s) are kept.

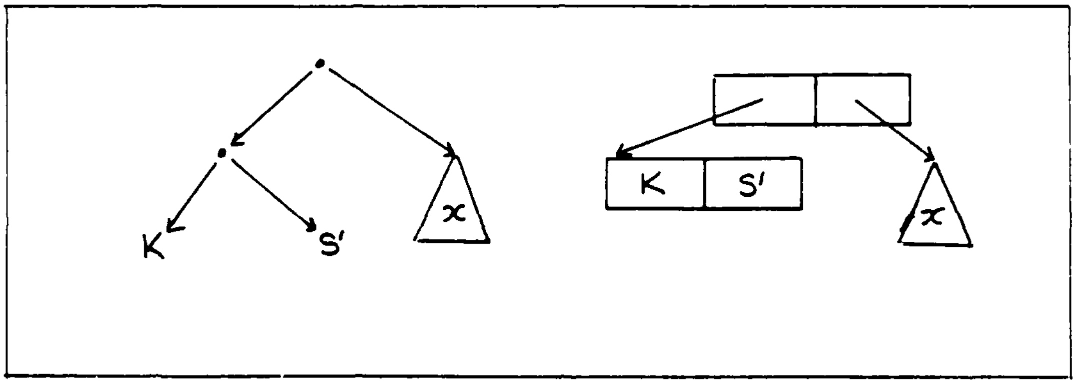
An LNF-wff combination **X**, having operator **OPR** and operand **OPD**, is represented in the machine by a CONS cell, the CAR of which points at the representation of **OPR** and the CDR of which points at the representation of **OPD**.

A CONS cell will be displayed as a rectangle divided in half -- the left half being the CAR and the right the CDR. Arrows are used to represent pointers. As in diagrams displaying LNF-wffs (see Chapter 2), labeled triangles will be used to abbreviate whole LNF-wff representations.

[1] LNF-of-wff simulates the LNF-M machine described in Chapter 2.

[2] The user specifies (via directives) how much reduction is to be performed.

[3] A session with LNF has been recorded and placed in Appendix D.



An LNF-wff and Its ZetaLisp Representation
Figure 3.1

The system function which builds the machine representation of a combination from the representations of its operator and operand is called Combine. From time to time ZetaLisp function definitions will be displayed. They always take the form:

(DEFUN Function-name (formal$_1$ $\cdots$ formal$_n$) body).

The simple definition of Combine follows:

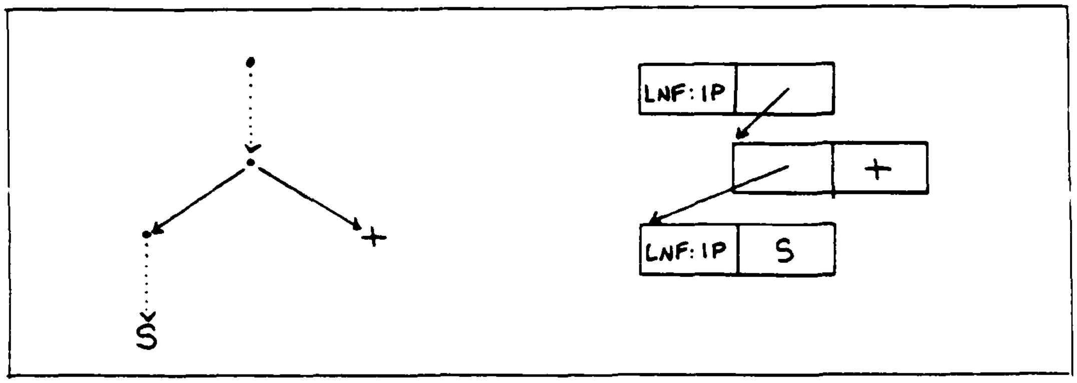
(DEFUN Combine (wff$_1$ wff$_2$) (CONS wff$_1$ wff$_2$)).

When displaying ZetaLisp code, ZetaLisp primitives (such as DEFUN and CONS) appear in uppercase, defined functions appear capitalized, and formal parameters appear in lowercase.

Recall from Chapter 2 the function COMBINE. Its domain was restricted to COMPATIBLE LNF-wffs. Since LNF-wffs are being represented by ZetaLisp objects, incompatible representations cannot exist — and therefore the implemention's ZetaLisp function (Combine) need not perform a compatibility check.

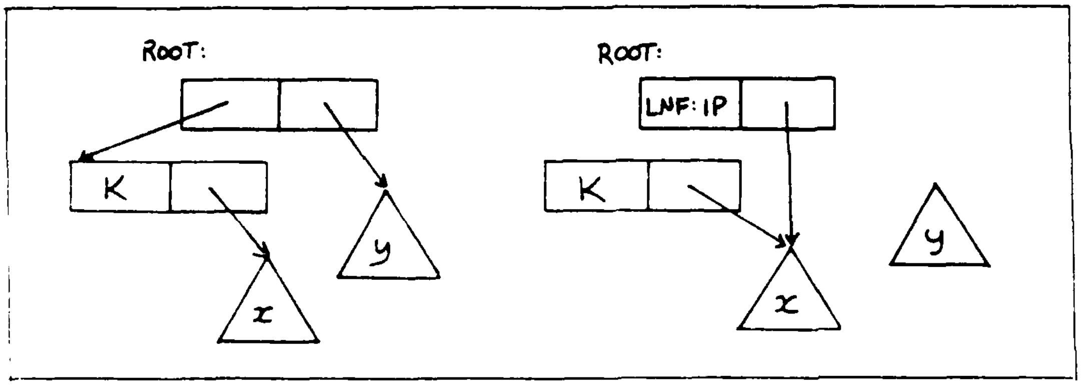
It remains to describe how forwarded vertices are represented. Forwarding vertices (like combinations) are also represented by CONS cells — the CAR of which is a flag (the ZetaLisp symbol LNF:IP[4]) telling the system that this is not a combination but a forwarding vertex. The CDR of the CONS cell points to the representation of the LNF-wff to which the vertex has been forwarded. An example follows:

[4] In ZetaLisp there is more than one namespace ZetaLisp symbols live in "packages" — and are written P S where P is the name of the package and S the name of the symbol Thus, the symbol LNF IP (IP for Invisible Pointer) lives in the LNF package — a private package inaccessible to the user of the LNF system There is no danger that the symbol LNF IP could be confused with a user constructor having the name IP as all user symbols are placed in the USER package The prefix "USER " is assumed by ZetaLisp if no prefix is provided



An LNF-wff and Its Representation
Figure 3.2

Recall from Chapter 2 that, in the LNF-calculus, only combinations are ever forwarded.[5] Given the representation above, combination forwarding may be accomplished by simply overwriting the representation's CAR (with the symbol LNF:IP) and CDR (with the pointer to the wff to which the combination is being forwarded). A representation of a K redex-reductum pair, illustrating combination forwarding, is displayed below.



An Illustration of Combination Forwarding
Figure 3.3

This method of combination forwarding is a modified version of the one presented in [Turner 1979c]. Turner, instead of marking the combination as having been forwarded, overwrote the combination's operator with the identity functor I and the operand with the wff to which the combination was being forwarded. LNF's implementation differs from Turner's here because it was felt that I redexes and forwarding vertices should be distinguishable.

3.3. Compiling LNF Expressions to LNF-wffs

LNF-wffs (even in a linearized format) are not "user friendly". The LNF language, defined below, attempts to satisfy the human need for a higher level of expression. An LNF program is an expression (LNF-exp). The system function Compile translates well-formed LNF-exps into LNF-wffs.

[5] Combinations are forwarded in the LNF-calculus by the function FORWARD-COMB

Please note that only well-formed LNF-exps are translated. No attempt has been made to implement input error handling — when presented with unrecognizable input the system simply stops. In the discussion to follow, therefore, it will be assumed that user input is always well-formed. Although LNF-exps are strings of characters, for purposes of discussion, an LNF-exp will be assumed to be an entity which wears its syntactic category on its sleeve and whose immediate constituents can be easily selected. That is to say, an LNF-exp's abstract syntax is what's important here, not its concrete syntax.[6]

The set of well-formed LNF expressions (LNF-exp) may be partitioned into five subsets. They are:

- Simple expressions (SIMPLE-exp)
- Lambda expressions (LAMBDA-exp)
- Expressions having auxiliary declarations (WITH-AUX-DECL-exp)
- List expressions (LIST-exp)
- Conditional expressions (IF-exp ∪ CASE-exp)

The transformation process which produces LNF-wffs from LNF-exps will now be detailed. The discussion of the process will be broken up by expression type. Each of the following subsections takes one LNF expression class and shows how expressions in that class are transformed.

3.3.1. Simple Expressions

Simple Expressions (SIMPLE-exps) are just linearized LNF-wffs with two exceptions.

The first exception is that atomic SIMPLE-exps may be variables as well as functors and constructors. All variable occurrences in LNF-exps are bound occurrences. Variables are distinguished from constructors and functors by their first character. All variables begin with the character "?". A variable is represented in the machine by the ZetaLisp symbol having the same name — just like contructors and functors.

The other exception is that parenthesized LNF-exps also fall into the class of SIMPLE-exp. Parentheses serve the same purpose in LNF-exps as they did in the SKI-calculus and in the linear representations of SKI-G-wffs and LNF-wffs, i.e. they are used for grouping only.

LNF's Compiler (from LNF-exps to LNF-wffs), as mentioned above, is implemented by a suite of functions; the topmost of which is called Compile. The (partial) ZetaLisp definition of Compile is:

[6] For those readers interested, a BNF-like description of LNF's concrete syntax may be found in Appendix B.

```
(DEFUN Compile (exp)
 (COND ((Atom-p exp) exp)
      ((Combination-p exp) (Combine (Compile (Operator exp))
                                    (Compile (Operand exp))))
      ((Parened-exp-p exp) (Compile (Exp-inside-parens exp)))
      ;; REST OF THE BRANCHES OF THE COND TO BE SUPPLIED
      ;; IN LATER SECTIONS OF THIS CHAPTER
      ))
```

The majority of functions making up the implementation will not be displayed. Many of the low level functions, such as the predicates Atomic-exp-p, Combination-p, and Parened-exp-p, and the selector functions Operator, Operand, and Exp-inside-parens, which don't provide much insight into the implementation, will not be presented.

Since it is now known how SIMPLE-exps are represented by the system, to illustrate how the more complex LNF-exps are compiled, it suffices to show how these other types of LNF-exps are translated into SIMPLE-exps.

3.3.2. Lambda Expressions

The LAMBDA-exps in LNF differ from abstractions in the λ-calculus. In the λ-calculus abstractions take the form

(λ v **X**), where v is a variable and **X** is a λ-wff

In LNF, however, a LAMBDA-exp takes the form

(λ (**BE**$_1$ ... **BE**$_n$) **BODY**), where
each **BE**$_i$ is a bound expression and **BODY** is an LNF-exp

Some LAMBDA-exps:

λ (?x) (+ ?x ?x)

λ ((vec ?x ?y) (vec ?w ?z)) (vec (+ ?x ?w) (+ ?y ?z))

λ (0) 1

λ (?x ?) ?x

The two differences between λ-calculus abstractions and LNF LAMBDA-exps are (1) a LAMBDA-exp can have more than one formal parameter while a λ-calculus abstraction has only one and (2) each formal parameter of a LAMBDA-exp can be a bound expression instead of being limited to a bound variable as in the case of the abstraction. The first difference may be easily discharged as the LAMBDA-exp:

(λ (**BE**$_1$ ... **BE**$_n$) **BODY**)

having n formal parameters is merely shorthand for the LAMBDA-exp

(λ (**BE**$_1$) (...(λ (**BE**$_n$) **BODY**)...)),

which has only one. Thus, a LAMBDA-exp possessing two formal parameters is not representing a binary function. It represents a unary function whose body is also a unary function — i.e. a second order function.

A bound expression (BE) is either a named variable (written 'name), an anonymous variable (written as ? alone), a constructed bound expression (CONSTRUCTED-BE), which is simply a construction whose arguments are BEs, or a list bound expression (LIST-BE) which is sugar for a CONSTRUCTED-BE. D.A. Turner, in his excellent paper: "A New Implementation Technique for Applicative Languages" ([Turner 1979c]), also extended the notion of formal parameters from simple variables. He limited his bound expressions, however, to being what this author is calling LIST-BEs — LIST-BEs being sugar for expressions of the form: PAIR **X** **Y**. LNF's BEs are simply Turner's pairs generalized to be arbitrary constructions.

Why have CONSTRUCTED-BEs been introduced into the language? A CONSTRUCTED-BE, acting as a formal parameter in a LAMBDA-exp, plays the part of an argument template[7]. A compiled LAMBDA-exp combined with (applied to) an argument will be reducible iff the argument matches the LAMBDA-exp's BE. An argument **A** *matches* a BE **B** iff

(*or* **B** is a variable (anonymous or named)
  (*and* **B** is a CONSTRUCTED-BE having the form: **c** $\mathbf{BE}_1 \cdots \mathbf{BE}_n$
    A has the lazy-normal form **c** $\mathbf{A}_1 \ \ \mathbf{A}_n$
    $\mathbf{A}_1$ matches $\mathbf{BE}_1$, . and $\mathbf{A}_n$ matches $\mathbf{BE}_n$)).

Formal parameters have been generalized from being only bound variables to include constructed bound expressions (CONSTRUCTED-BEs) for two pragmatic reasons:

1. CONSTRUCTED-BEs obviate the need for many user defined selector functions. As an example, consider the function which performs vector addition. Using CONSTRUCTED-BEs, the LAMBDA-exp is written:
   λ ((vec ?x ?y) (vec ?w ?z)) (vec (+ ?x ?w) (+ ?y ?z))
   Without the use of the CONSTRUCTED-BEs, the LAMBDA-exp becomes:
   λ (?u ?v) (vec (+ (xc ?u) (xc ?v)) (+ (yc ?u) (yc ?v)))
   where xc (yc) is the selector function which extracts the x (y) component of a vector.

2. A CONSTRUCTED-BE ensures that its LAMBDA-exp is used for arguments of the kind the user intended — i.e. arguments which match the template. In the above example, the CONSTRUCTED-BEs in the formal parameter list of the first LAMBDA-exp guarantee the function is used only with vectors. No such guarantee is provided by the formal parameters of the second LAMBDA-exp.

An important question remains. How is (BE to argument) matching performed after all of the variables have been abstracted away by the compiler? The compiler (the function Compile) must produce, from a LAMBDA-exp having the formal parameter **BE**, an LNF-wff (in which there are no variables) which is capable of checking if the argument to which it is being applied would have matched **BE**. This is accomplished by the generalized abstraction algorithm Abstract-be, which makes use of the functor: A-S (standing for Abstract Structure), in addition to the functors used in the definition of C-T-

[7] Both anonymous and named variables also act as templates — templates that will match any argument.

ABS (Curry's and Turner's abstraction algorithm used in the LNF-calculus).

The COND-branch in the ZetaLisp definition of the Compile function which deals with LAMBDA-exps is as follows:

```
...
((Lambda-exp-p exp)
 (Abstract-each-be (Formals exp) (Compile (Body exp))))
...
```

Recall the definition of λ-TO-LNF (from Chapter 2) which translated λ-wffs into LNF-wffs. The program section above mirrors the first part of the definition of λ-TO-LNF repeated below·

```
(if X = (λ v B)
  then C-T-ABS[v,λ-TO-LNF[B]]
  ...
```

The definition of Abstract-each-be:

```
(DEFUN Abstract-each-be (non-empty-be-list compiled-body)
  (LET ((compiled-be ;; BE
        (Compile (Last-be-in-list non-empty-be-list))))
    ;; IN
    (IF (Only-one-be-in non-empty-be-list)
        ;; THEN
        (Abstract-be compiled-be compiled-body)
        ;; ELSE
        (Abstract-each-be
          (All-but-last-in non-empty-be-list)
          (Abstract-be compiled-be compiled-body)))))
```

In addition to being able to abstract simple variables, Abstract-be must be able to abstract away anonymous variables and constructed bound expressions. Note that in the definition of Abstract-each-be (above) the BEs are compiled before being passed as arguments to Abstract-be. LIST-BEs are transformed into CONSTRUCTED-BEs (having the form: (PAIR **X Y**)) by this process. The ZetaLisp definitions of Abstract-be and its helper function A-S-or-A-S′-comb[8] come next:

[8] The functor A-S′ is used when abstracting away variables introduced in CASE-exps

```
(DEFUN Abstract-be
  (be compiled-body &optional (arg-reduced-p NIL))
  ;; IF THIRD ARG NOT PROVIDED THEN IT TAKES ON VALUE NIL
  (COND ((Anonymous-variable-p be) (Combine 'K compiled-body))
       ((Named-variable-p be) (C-T-abs[9] be compiled-body))
       (T ;; be is a desugared CONSTRUCTED-BE — i.e.
          ;; a construction whose arguments are BEs
        (A-S-or-A-S'-comb
          arg-reduced-p
          (Constructor be)
          (Number-of-args be)
          (Abstract-each-be (Args be) compiled-body)))))

(DEFUN A-S-or-A-S'-comb (use-prime-p c n Inf-wff)
  (Combine
    (Combine (Combine (IF use-prime-p 'A-S' 'A-S) c) n)
    Inf-wff))
```

Some examples of LAMBDA-exps and their SIMPLE-exp equivalents:

```
λ (?x) (+ ?x ?x)
W +

λ ((pair ?x ?y)) (+ ?x ?y)
A-S PAIR 2 +

λ ([?x•?y]) (+ ?x ?y)
A-S PAIR 2 +

λ ((vec ?x ?y) (vec ?w ?z)) (vec (+ ?x ?w) (+ ?y ?z))
A-S VEC 2 (C' (B' (A-S VEC 2)) (B' C (B' B VEC) +) +)

λ (?u ?v) (vec (+ (xc ?u) (xc ?v)) (+ (yc ?u) (yc ?v)))
S' S (C (B' (B' VEC) + XC) XC) (C (B' B + YC) YC)

λ ((tree ?l ? ?r)) (append (flatten ?l) (flatten ?r))
A-S TREE 3 (B K (C (B' B APPEND FLATTEN) FLATTEN))

λ (0) 1
A-S 0 0 1

λ (?x ?) ?x
K
```

A step by step look at on: of the more complex sample transformations follows. Starting with:

[9] ZetaLisp version of the function C-T-ABS presented at the end of Chapter 2

λ ((vec ?x ?y) (vec ?w ?z)) (vec (+ ?x ?w) (+ ?y ?z))

First, the BE:

(vec ?w ?z)

is abstracted from the body:

(vec (+ ?x ?w) (+ ?y ?z))

yielding:

A-S VEC 2 (C (B′ B vec (+ ?x)) (+ ?y)).

Now the BE:

(vec ?x ?y)

is abstracted from:

A-S VEC 2 (C (B′ B vec (+ ?x)) (+ ?y)).

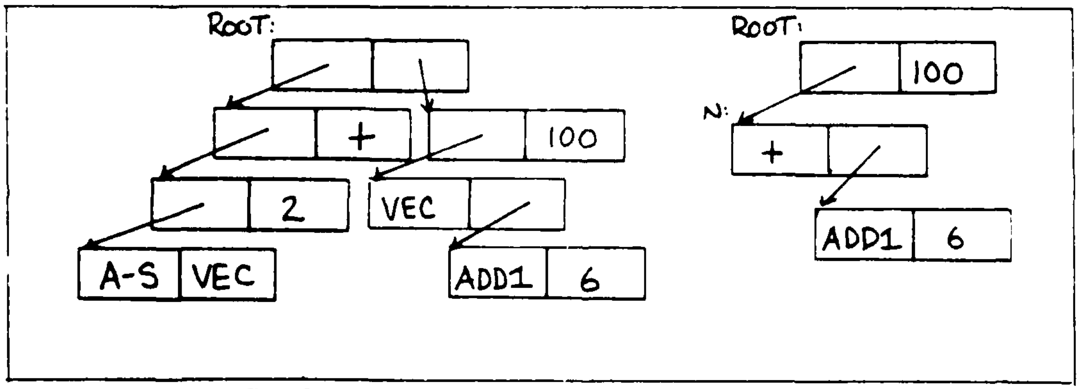
The result is the LNF-wff:

A-S VEC 2 (C′ (B′ (A-S VEC 2)) (B′ C (B′ B VEC) +) +).

The adventurous reader may wish to verify that the other sample compilations have been performed properly.

This compiled expression will now be applied to arguments and reduced to lazy-normal form. To make sense of the reduction, one must know the rules for the functors involved. The rules for the functor A-S (originally presented in Chapter 2) are repeated below — it is assumed that rules for the now familiar functors: B, B′, C, C′, and + need not be redisplayed.

$\text{A-S}\ \mathbf{c\ i\ X}\ (\mathbf{c\ Z}_1 \cdots \mathbf{Z}_i) \rightarrow \mathbf{X\ Z}_1 \cdots \mathbf{Z}_i$
$\text{A-S}\ \mathbf{c\ i\ X\ RDU} \rightarrow \text{A-S}\ \mathbf{c\ i\ X\ IMR}$



ZetaLisp representation of an A-S reduction
Figure 3.4

The function: A-S VEC 2 (C′ (B′ (A-S VEC 2)) (B′ C (B′ B VEC) +) +) applied to arguments (VEC 10 20) and (VEC 30 40) reduces first to:

C′ (B′ (A-S VEC 2)) (B′ C (B′ B VEC) +) + 10 20 (VEC 30 40),

then to:

B′ (A-S VEC 2) (B′ C (B′ B VEC) + 10) + 20 (VEC 30 40),

then to:

A-S VEC 2 (B′ C (B′ B VEC) + 10 (+ 20)) (VEC 30 40),

then to:

B′ C (B′ B VEC) + 10 (+ 20) 30 40,

then to:

C (B′ B VEC (+ 10)) (+ 20) 30 40,

then to:

B′ B VEC (+ 10) 30 (+ 20) 40

then to:

B (VEC (+ 10 30)) (+ 20) 40

and finally to:

VEC (+ 10 30) (+ 20 40)

which, because it is a construction, is in lazy-normal form.

The combination labeled "N:" is a newly created combination.

It was mentioned above that Turner, in [Turner 1979c], had allowed formal parameters to be pairs (and pairs of pairs etc.) as well as simple variables. His abstraction algorithm, when it had the task of abstracting a formal parameter of the form PAIR **HD** **TL** from an expression **EXP**, produced a combination of the form:

U abstract[**HD**,abstract[**TL**,**EXP**]]

where the functor U (standing for Unpair) was characterized by the two rules:

U **Z** (PAIR **X** **Y**) → **Z** **X** **Y** and
U **Z** **RDU** → U **Z** **IMR**.

Note that the function yielded by Turner's algorithm: (U **FN**) behaves identically to the function (A-S PAIR 2 **FN**) — the function that Abstract-be would have produced in this situation. It can be seen that Turner's functor U is the instance of the function (A-S **c** **n**) where **c** has been instantiated with the constructor PAIR and **n** with 2.

3.3.3. Expressions with Auxiliary Declarations

Expressions having auxiliary declarations come in three flavors: WHERE-exps, WHERE*-exps, and WHEREREC-exps. Each of these three types of expression is a variable binding form which, unlike LAMBDA-exps, associates expressions with the variables introduced.[10]

[10] Other FP languages possess equivalent forms which introduce the variable before its use. These forms are usually initiated by the keywords: LET, LET*, and LETREC.

EXAMPLES: (of WHERE, WHERE*, and WHEREREC expressions)

```
(+ ?x ?y) where ?x = 3 & ?y = 4

(thrice double 5) where
  thrice ?f ?x = ?f (?f (?f ?x)) &
  double ?x = × 2 ?x

(+ ?x ?y) where (tree ?x ? ?y) = some-tree

(× ?x ?y) where* ?x = 3 ; ?y = (factorial ?x)

?p1 whererec ?p1 = [1•?p2] & ?p2 = [2•?p1]

(factorial 10) whererec
  factorial ?n = (if (zerop ?n) then 1
                    else (× ?n (factorial (sub1 ?n))))

(app [1,2,3] list) whererec
  {app [ ] ?z = ?z |
   app [?x•?r] ?z = [?x•(app ?r ?z)]}
```

The three expression types differ from one another by the different scopes given to the introduced variables. For example consider the scope of the variables in the bound expression $\mathbf{be}_2$ in each of the following three expressions, where: **exp**[11], $\mathbf{e}_1$, $\mathbf{e}_2$, and $\mathbf{e}_3$ are LNF-exps and $\mathbf{be}_1$, $\mathbf{be}_2$, and $\mathbf{be}_3$ are bound expressions:

exp WHERE $\mathbf{be}_1 = \mathbf{e}_1$ & $\mathbf{be}_2 = \mathbf{e}_2$ & $\mathbf{be}_3 = \mathbf{e}_3$
exp WHERE* $\mathbf{be}_1 = \mathbf{e}_1$; $\mathbf{be}_2 = \mathbf{e}_2$; $\mathbf{be}_3 = \mathbf{e}_3$
exp WHEREREC $\mathbf{be}_1 = \mathbf{e}_1$ & $\mathbf{be}_2 = \mathbf{e}_2$ & $\mathbf{be}_3 = \mathbf{e}_3$

In the first expression, the scope of the variables occurring in $\mathbf{be}_2$ is **exp** alone; in the second their scope is **exp** and $\mathbf{e}_3$; and in the third their scope is **exp**, $\mathbf{e}_1$, $\mathbf{e}_2$, and $\mathbf{e}_3$. Note the use of semicolons as separators in the WHERE*-exp. Semicolons have been used to suggest a sequence. In WHERE*-exps, the scope of $\mathbf{be}_2$'s variables includes, besides the main expression, the definiens of any succeeding declarations — thus the ordering of the declarations is important in WHERE*-exps. The ordering of the declarations in WHERE-exps and WHEREREC-exps is not important; hence the use of ampersand as a separator between their declarations. Function declarations like:

```
thrice ?f ?x = ?f (?f (?f ?x))
```

and

```
{app [ ] ?z = ?z |
 app [?x•?r] ?z = [?x•(app ?r ?z)]}
```

are transformed into declarations of the form: ?function-name = LNF-exp.[12] Hence function declarations, even though they differ in outward appearance, may be compiled, after this transformation, like any other declaration. It will now be shown how each of the

[11] The expression exp is called the main expression in these constructs.

[12] This transformation will be detailed below.

three types of expressions having auxiliary declarations is transformed into an equivalent simple expression

3.3.3.1. WHERE-exps

A WHERE-exp having only one declaration, is sugar for a combination having an operator which is a LAMBDA-exp — i.e. a β-redex. The WHERE-exp

exp WHERE **be** = **e**

is a disguised form of the combination

((λ (**be**) **exp**) **e**)

A WHERE-exp having more than one declaration also has a SIMPLE-exp equivalent which is a combination. Recall that its declarations are mutually independent and have only the main expression as their scope. Therefore, the WHERE-exp.

exp WHERE $\mathbf{be}_1 = \mathbf{e}_1$ & $\mathbf{be}_2 = \mathbf{e}_2$ & $\mathbf{be}_3 = \mathbf{e}_3$

may be seen as sugar for the combination

(λ ($\mathbf{be}_1$) (λ ($\mathbf{be}_2$) (λ ($\mathbf{be}_3$) **exp**))) $\mathbf{e}_1$ $\mathbf{e}_2$ $\mathbf{e}_3$

It is easy to see that the scope of each of the $\mathbf{be}_i$'s is just the main expression of the WHERE-exp **exp**.

As a concrete example, consider the WHERE-exp:

(+ ?x ?y) where ?x = 3 & ?y = 4.

Its SIMPLE-exp equivalent is the combination

(λ (?x) (λ (?y) (+ ?x ?y))) 3 4

which compiles to the LNF-wff

+ 3 4.

Although it appears that the compiler has performed two β contractions, this is not the case. In fact, what the compiler (specifically, the ZetaLisp function C-T-abs) has done has been to make use of the equivalence between the LAMBDA-exp: (λ (?x) (M ?x)) and the expression: M, which holds when ?x does not occur in M.

3.3.3.2. WHERE*-exps

A WHERE*-exp might be called sugarcoated sugar, for it is sugar for a telescoped WHERE-exp. For example, the abstract WHERE*-exp:

exp WHERE* $\mathbf{be}_1 = \mathbf{e}_1$; $\mathbf{be}_2 = \mathbf{e}_2$; $\mathbf{be}_3 = \mathbf{e}_3$

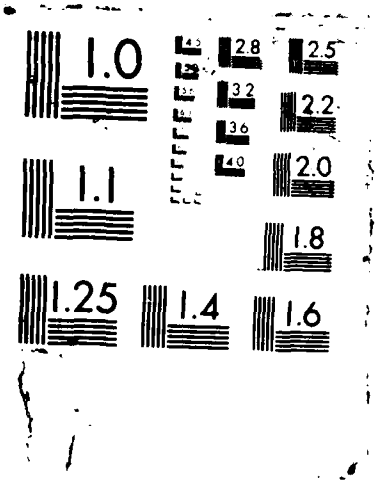
is syntactic sugar for this WHERE-exp:

((**exp** WHERE $\mathbf{be}_3 = \mathbf{e}_3$) WHERE $\mathbf{be}_2 = \mathbf{e}_2$) WHERE $\mathbf{be}_1 = \mathbf{e}_1$

which, in turn, is sugar for the combination:

AD-A190 166 NEW GENERATION KNOWLEDGE PROCESSING VOLUME 2(U) 2/3
SYRACUSE UNIV NY J A ROBINSON ET AL. OCT 87
RADC-TR-87-165-VOL-2 F30602-84-K-0001

UNCLASSIFIED F/G 12/5 NL


1.0
2.8
2.5
3.2
2.2
3.6
4.0
2.0
1.1
1.8
1.25
1.4
1.6

((λ (**be**$_1$) ((λ (**be**$_2$) *((λ (be$_3$) exp) e$_3$)*) **e**$_2$)) **e**$_1$).

The scope of **be**$_2$ has been italicized to illustrate that its scope really is **exp** and **e**$_3$ as was claimed. Note that a WHERE*-exp having n declarations, when desugared, contains (at least) n β-redexes. Note also that a WHERE*-exp, having main expression **e** and a single declaration **d**, is compiled identically to the WHERE-exp having main expression **e** and the single declaration **d**.

As a concrete example, the WHERE*-exp:

(+ ?x ?y) where* ?x = 3 ; ?y = (factorial ?x)

is sugar for the WHERE-exp:

((+ ?x ?y) where ?y = (factorial ?x)) where ?x = 3

which is sugar for the combination:

((λ (?x) ((λ (?y) (+ ?x ?y)) (factorial ?x))) 3).

The WHERE*-exp, the WHERE-exp, and the combination thus compile to the same LNF-wff:

S - factorial 3.

3.3.3.3. WHEREREC-exps

The declarations in a WHEREREC-exp are neither sequential (like those in WHERE*-exps) nor mutually independent (like the ones in WHERE-exps), but are mutually dependent. That is to say that the scope of each definiendum includes all of the definientia in addition to the main expression. Just like WHERE*-exps and WHERE-exps, however, WHEREREC-exps can be desugared into simple expressions. Before showing how to desugar a WHEREREC-exp having many declarations, it will be shown how to desugar a WHEREREC-exp having only one declaration. Consider the WHEREREC-exp:

exp WHEREREC **be** = **ebe**,

where **ebe** is an LNF-exp containing some free occurrences of the variables in **be**. The following combination is equivalent to **ebe**:

(λ (**be**) **ebe**) **be**.

This combination also has the property that its operator does not contain any free occurrences of the variables in **be**. Replacing **ebe** with ((λ (**be**) **ebe**) **be**) in **be**'s declaration gives a declaration having the form:

be = **F be**,

where no variable in **be** occurs free in the function **F**. Any fixed-point of the function **F** (having a form which matches **be**) will satisfy this equation.[13] Recall from Chapter 2 that the combination (Y **G**) is equal to (**G** (Y **G**)) for all functions **G**. Thus (Y **G**) is a fixed-point of any function **G**. Hence (Y **F**) is a fixed-point of the function

[13] All fixed-points of **F** will be of this form since, by its definition, it is only applicable to arguments of the desired form.

F = λ (**be**) **ebe**.

Therefore, the noncircular declaration:

be = Y (λ (**be**) **ebe**)

is equivalent to the circular one in the WHEREREC-exp. Since the declaration isn't circular, the WHEREREC-exp may be desugared (just like a WHERE-exp) into the combination:

(λ (**be**) **exp**) (Y (λ (**be**) **ebe**)).

A concrete example follows. The WHEREREC-exp:

(first 5 ?x) whererec ?x = [1,2•?x]

is tranformed first to:

(first 5 ?x) whererec ?x = ((λ (?x) [1,2•?x]) ?x)

and then to:

(first 5 ?x) where ?x = (Y (λ (?x) [1,2•?x]))

and finally to:

(λ (?x) (first 5 ?x)) (Y (λ (?x) [1,2•?x])).

This combination is then compiled to the LNF-wff:

FIRST 5 (Y (B (PAIR 1) (PAIR 2))).

Another example, whose definiendum is a CONSTRUCTED-BE, follows:

?x whererec [?x•?y] = [[1•?y]•[2•?x]]

is transformed first to:

?x whererec [?x•?y] = (λ ([?x•?y]) [[1•?y]•[2•?x]]) [?x•?y]

then to:

?x where [?x•?y] = (Y (λ ([?x•?y]) [[1•?y]•[2•?x]]))

and finally to:

(λ ([?x•?y]) ?x) (Y (λ ([?x•?y]) [[1•?y]•[2•?x]])).

The function Compile would now dictate that this combination be compiled to:

```
A-S
 PAIR
 2
 K
 (Y (A-S PAIR 2 (B (C′ PAIR (PAIR 1)) (PAIR 2)))).
```

This LNF-wff, however, has no lazy-normal form! To see this, recall the rules characterizing the functor A-S:

A-S **c i X** (**c** $\mathbf{Z}_1 \cdots \mathbf{Z}_i$) → **X** $\mathbf{Z}_1 \cdots \mathbf{Z}_i$
A-S **c i X RDU** → A-S **c i X IMR**

The functor A-S's second rule says that A-S's fourth argument must be reduced before the first rule can be applied — i.e. any function having the form (A-S **c i X**) is strict. Hence to reduce the LNF-wff produced by the compiler, one must first reduce its fourth argument. Its fourth argument has the form: (Y **G**), where **G** is also a strict function. Since this combination reduces to (**G** (**G** ...)), it should be clear that **G** being strict implies that this combination will not have a lazy-normal form. Therefore, the original LNF-wff will not have a lazy-normal form.

To solve this problem — that is, to compile the WHEREREC-exp to an LNF-wff which has a lazy-normal form — the strict function:

A-S PAIR 2 (B (C′ PAIR (PAIR 1)) (PAIR 2))

is replaced by an equivalent (in this context) nonstrict function. The function which is used in its place is:

APP-TO-ARGS 2 (B (C′ PAIR (PAIR 1)) (PAIR 2)).

Recall from Chapter 2 the reduction rule which characterizes the functor APP-TO-ARGS:

APP-TO-ARGS **i X Y** → **X** (ARG 1 **Y**) ... (ARG **i Y**).

This rule implies that any function of the form (APP-TO-ARGS **i X**) is nonstrict (it doesn't care what form its argument **Y** takes) and, when applied to an LNF-wff having the form $(\mathbf{c}\ \mathbf{Z}_1 \cdots \mathbf{Z}_i)$, reduces to the same LNF-wff to which the combination (A-S **c i X** $(\mathbf{c}\ \mathbf{Z}_1 \cdots \mathbf{Z}_i)$) reduces. To see this, return to the sample LNF-wff (having made the function replacement) and view a linearized display of its reduction to lazy-normal form.

A-S PAIR 2 K (Y (APP-TO-ARGS 2 (B (C′ PAIR (PAIR 1)) (PAIR 2))))

reduces to:

A-S PAIR 2 K (APP-TO-ARGS 2 (B (C′ PAIR (PAIR 1)) (PAIR 2)) **H**)

via the Y rule, where **H** is the cyclic LNF-wff:

(APP-TO-ARGS 2 (B (C′ PAIR (PAIR 1)) (PAIR 2)) **H**).

The next reduction, using APP-TO-ARGS' rule, yields:

A-S PAIR 2 K (B (C′ PAIR (PAIR 1)) (PAIR 2) (ARG 1 **H**) (ARG 2 **H**)),

which via the B rule becomes:

A-S PAIR 2 K (C′ PAIR (PAIR 1) (PAIR 2 (ARG 1 **H**)) (ARG 2 **H**)),

which reduces via the C′ rule to:

A-S PAIR 2 K (PAIR (PAIR 1 (ARG 2 **H**)) (PAIR 2 (ARG 1 **H**))).

Finally, A-S's first rule may be applied. The result is:

K (PAIR 1 (ARG 2 **H**)) (PAIR 2 (ARG 1 **H**))

which reduces via the rule for K to the construction (a pair):

PAIR 1 (ARG 2 **H**),

which is in lazy-normal form.

Before continuing on with WHEREREC-exps, it might be mentioned that Turner in [Turner 1979c], when presenting his compilation scheme for expressions with mutually dependent declarations, made the error of using his strict functor U instead of a nonstrict equivalent. The functor he meant to use ([Turner 1983]), instead of U, was the nonstrict functor U′ characterized by the rule:

$$\mathrm{U}' \; \mathbf{X} \; \mathbf{Y} \rightarrow \mathbf{X} \; (\mathrm{HD} \; \mathbf{Y}) \; (\mathrm{TL} \; \mathbf{Y})$$

where HD and TL are the selector functions which retrieve the head and tail of a pair, respectively. This functor U′ may be viewed as APP-TO-ARGS restricted to working on pairs — with HD and TL playing the parts of the functions (ARG 1) and (ARG 2).

Up to this point, the WHEREREC-exps that have been dealt with have contained only one declaration. WHEREREC-exps having more than one declaration are compiled by first transforming them into an equivalent WHEREREC-exp having only one declaration, and then compiling this new WHEREREC-exp as detailed above. Consider the WHEREREC-exp below:

$$\mathbf{exp} \; \mathrm{WHEREREC} \; \mathbf{be}_1 = \mathbf{e}_1 \; \& \; \mathbf{be}_2 = \mathbf{e}_2 \; \& \; \mathbf{be}_3 = \mathbf{e}_3$$

having three declarations. The following WHEREREC-exp, having only one declaration, is equivalent to it:

$$\mathbf{exp} \; \mathrm{WHEREREC} \; (\mathrm{OPDS} \; \mathbf{be}_1 \; \mathbf{be}_2 \; \mathbf{be}_3) = \; (\mathrm{OPDS} \; \mathbf{e}_1 \; \mathbf{e}_2 \; \mathbf{e}_3),$$

where OPDS is simply a constructor. Since it has just been shown how to compile WHEREREC-exps of this form, nothing else need be said.

As a concrete example, consider the WHEREREC-exp:

?p1 whererec ?p1 = [1•?p2] & ?p2 = [2•?p1].

This expression is transformed to the equivalent WHEREREC-exp:

?p1 whererec (OPDS ?p1 ?p2) = (OPDS [1•?p2] [2•?p1])

which is equivalent to:

```
?p1 whererec (OPDS ?p1 ?p2) =
            (λ ((OPDS ?p1 ?p2)) (OPDS [1•?p2] [2•?p1])) (OPDS ?p1 ?p2)
```

which is equivalent to the WHERE-exp:

?p1 where (OPDS ?p1 ?p2) = Y (λ ((OPDS ?p1 ?p2)) (OPDS [1•?p2] [2•?p1])).

This WHERE-exp is just sugar for the β-redex:

(λ ((OPDS ?p1 ?p2)) ?p1) (Y (λ ((OPDS ?p1 ?p2)) (OPDS [1•?p2] [2•?p1])))

which compiles to the LNF-wff:

```
A-S
 OPDS
 2
 K
 (Y (APP-TO-ARGS 2 (B (C′ OPDS (PAIR 1)) (PAIR 2)))).
```

In each of the four FP languages:

- SASL — St. Andrews Static Language ([Turner 1979b] and [Turner 1979c]),
- KRC — Kent Recursive Calculator ([Turner 1981a], [Turner 1981b], and [Turner 1982]),
- Miranda — D.A. Turner's most recent effort ([Turner 1984b]), and
- ARC SASL — developed by Burroughs Corporation in close collaboration with D.A. Turner ([Richards 1984])

there is only one expression form having auxiliary declarations. Each of these languages has collapsed the WHERE, WHERE*, and WHEREREC expressions into one expression: the WHERE expression. The compiler detects which definientia are dependent on which other declarations and then compiles the WHERE expression either like LNF's WHERE-exp, if the declarations are mutually independent, or like LNF's WHEREREC-exp, if any two declarations are found to be dependent. Some examples of this type of WHERE expression and their LNF equivalents follow.

```
KRC: x+y where x = 4*y; y = 2
LNF: + ?x ?y where* ?y = 2; ?x = × 4 ?y

KRC: p1 where p1 = 1:p2 ; p2 = 2:p1
LNF: ?p1 whererec ?p1 = [1•?p2] & ?p2 = [2•?p1]
```

Although many of LNF's constructs have been borrowed from Turner's languages, it was felt that Turner's WHERE construct was carrying too heavy a load. A reader of a KRC program must look inside each of the declarations in order to determine how the declarations interact. In LNF, however, the construct's keyword (either where, where*, or whererec) tells the reader whether the declarations are to be interpreted independently, sequentially, or mutually dependently. For this reason, it was decided to spread the work of Turner's WHERE expression appropriately to the WHERE, WHERE*, and WHEREREC expressions.

3.3.3.4. Function Declarations

Functions defined by an equation or a set of equations are both natural to write and easy to read and understand. It is assumed that, when a function is defined by a set of equations, the equations are pairwise independent — i.e. only one equation is applicable in any one situation. This property may be verified at compile time by attempting to unify ([Robinson 1965]) each pair of formal parameter lists. If a pair does unify, then the set of equations is not pairwise independent and therefore not suitable as a definiens for a deterministic function. The LNF compiler performs this check and issues a warning that the set of equations is "order dependent" if it finds a unifiable pair of formal parameter lists.

An example of an unacceptable equation set:

```
{factorial 0 = 1 | factorial ?n = × ?n (factorial (sub1 ?n))}
```

since ?n and 0 unify. The following definition of the list appending function:

{app [ ] ?z = ?z | app [?x•?r] ?z = [?x•(app ?r ?z)]}

is acceptable because there is no substitution (unifier) which will unify [ ] and [?x•?r].

It was claimed above that functions declared via a single equation like:

thrice ?f ?x = ?f (?f (?f ?x))

or by a set of equations like:

{app [ ] ?z = ?z | app [?x•?r] ?z = [?x•(app ?r ?z)]}

could be transformed into declarations of the form:

?function-name = **exp**.

This transformation will now be detailed.

First, consider a function declared via a single equation. These declarations take the form:

ZETALISP-ATOM **be**$_1$ ··· **be**$_n$ = **exp**.

An equation of this form is tranformed into the equivalent simple declaration:

?ZETALISP-ATOM = λ (**be**$_1$ ··· **be**$_n$) **exp**.[14]

For example, the equation:

thrice ?f ?x = ?f (?f (?f ?x))

is transformed into the declaration:

?thrice = λ (?f ?x) (?f (?f (?f ?x))).

As a concrete example, the WHERE-exp containing two function declarations:

```
(thrice double 5) where
  thrice ?f ?x = ?f (?f (?f ?x)) &
  double ?x = × 2 ?x
```

compiles to the LNF-wff:

C' C 5 (W (W B')) (× 2).

If the function is declared by a set of equations, then the equation set is transformed into a declaration of the form: ?function-name = **exp**, where **exp** is a LAMBDA-exp having a CASE-exp for a body. Consider as an example the following set of equations defining the function F:

{F **be**$_{11}$ **be**$_{12}$ = **body**$_1$ |
F **be**$_{21}$ **be**$_{22}$ = **body**$_2$ |
F **be**$_{31}$ **be**$_{32}$ = **body**$_3$}.

Note that for this set to yield a deterministic definition for the function F, no pair of

[14] Note that ?ZETALISP-ATOM must be substituted for (free occurrences of) ZETALISP-ATOM throughout the scope of the declaration. This scope varies depending on the type of expression (WHERE, WHERE•, or WHEREREC) of which the declaration is a part.

formal argument lists: $(\mathbf{be}_{i1}\ \mathbf{be}_{i2})$, $(\mathbf{be}_{j1}\ \mathbf{be}_{j2})$, $1 \leq i, j \leq 3$ & $i \neq j$ may be unifiable.

This equation set is sugar for the single equation:

```
F $\mathbf{v}_1$ $\mathbf{v}_2$ =
  case (opds $\mathbf{v}_1$ $\mathbf{v}_2$) in
    (opds $\mathbf{be}_{11}$ $\mathbf{be}_{12}$) → $\mathbf{body}_1$ |
    (opds $\mathbf{be}_{21}$ $\mathbf{be}_{22}$) → $\mathbf{body}_2$ |
    (opds $\mathbf{be}_{31}$ $\mathbf{be}_{32}$) → $\mathbf{body}_3$
  endcase
```

where $\mathbf{v}_1$ and $\mathbf{v}_2$ are two new system generated variables. This single equation is then transformed into a simple declaration using the method described above. The CASE-exp's transformation to a SIMPLE-exp is detailed in an upcoming section.

In certain situations, equation sets are transformed by the compiler into more efficiently reducible forms. In the case where the first parameters of the equations ($\mathbf{be}_{11}$, $\mathbf{be}_{21}$, and $\mathbf{be}_{31}$) are found to be pairwise independent (not unifiable), then the equation set is transformed to this equation:

```
F $\mathbf{v}_1$ =
  case $\mathbf{v}_1$ in
    $\mathbf{be}_{11}$ → (λ ($\mathbf{be}_{12}$) $\mathbf{body}_1$) |
    $\mathbf{be}_{21}$ → (λ ($\mathbf{be}_{22}$) $\mathbf{body}_2$) |
    $\mathbf{be}_{31}$ → (λ ($\mathbf{be}_{32}$) $\mathbf{body}_3$)
  endcase
```

which avoids the introduction of the variable $\mathbf{v}_2$ and the constructor opds; both of which add to the size of the code and in turn increase the number of reductions required anytime the function is used. The user of the system is therefore encouraged to place the "deciding" parameter (if one does exist) in the first parameter position. To illustrate the difference that the ordering of the formal parameters can make in the compiled code, observe the code produced for the following two equation sets. Both sets define a predicate accepting a number **n** and a list **l** as arguments and yielding TRUE iff **n** = length **l**. Their only difference is that the first predicate expects the number as first argument and the list as second and the secon; predicate expects them in reverse order. The first set:

```
{P1 ?n [?•?r] = P1 (sub1 ?n) ?r |
 P1 ?n [ ] = zerop ?n}
```

compiles to code containing 35 system generated functors, and to reduce the expression: P1 4 [1,2,3] to FALSE takes 79 reduction steps. The second set.

```
{P2 [?•?r] ?n = P2 ?r (sub1 ?n) |
 P2 [ ] ?n = zerop ?n}
```

compiles to code having only 17 new functors, and to reduce the expression P2 [1,2,3] 4 to FALSE takes only 38 reduction steps.

3.3.4. List Expressions

List expressions (expressions whose lazy-normal forms are either [ ] or take the shape: PAIR **X Y**) come in several flavors: (1) explicit lists like:

[1,2,3,4],

[flat,2,tire,1•23],

[a•b], and

[a,b,c•[ ]];

(2) arithmetic sequences like:

[1,..],

[10,10,..],

[1,3,..],

[0,-1,..],

[2,4,..,100],

[1,..,1000], and

[10,7.5,..,0]

and (3) implicit lists. Turner introduced implicit lists — he calls them "ZF expressions" — in his language KRC. He gave them this name since they are based on Zermelo-Frankel set abstraction — that is for every set **A** and predicate **P**, there is another set (**B**) whose members are exactly those members of **A** for which **P** holds. The equation defining the new set **B** is written in [Halmos 1974] as:

$$\mathbf{B} = \{\mathbf{x}\in\mathbf{A} : \mathbf{P}(\mathbf{x})\}.$$[15]

Implicit lists may be expressed in LNF in two ways. The first form is very similar to that used by Turner. The only difference is that, in LNF, square brackets have replaced curly braces as the construct's enclosing delimiters. Since these expressions really are lists and not sets — i.e. their lazy-normal form is either the empty list ([ ]) or a pair — it was felt that braces were inappropriate bits of sugar. A few examples of implicit lists, using the modified Turner syntax, follow:

[(subl (× 10 ?x)) | ?x∈[1,..,100]]

[[?x•?y] | ?x∈[1,..,5]; (odd ?x); ?y∈[100,101]]

[(+ ?x ?y) | [?x•?y]∈(zip [1,..,10] [100,..,110]); zerop (rem ?y ?x)]

A new syntax for implicit lists, which the author prefers over the one just described, exists in LNF. The essential differences between the two notations are: (1) where the local variables are introduced, and (2) the physical location of the scopes of the introduced variables. In the modified Turner syntax, variables are bound after their use (similar to WHERE constructs) and their scopes are not contiguous. In the new syntax, variables are bound before their use (similar to LET constructs) and scopes are always

[15] **Turner would have written this expression as {x|x∈A;P x}**

contiguous. The implicit lists above are redisplayed below using this new syntax:

```
for-each ?x∈[1,..100]
  instantiate (sub1 (× 10 ?x))

for-each ?x∈[1,..,5]
  such-that (odd ?x)
  and-for-each ?y∈[100,101]
  instantiate [?x•?y]

for-each [?x•?y]∈(zip [1,..,10] [100,..,110])
  such-that (zerop (rem ?y ?x))
  instantiate (+ ?x ?y)
```

The SIMPLE-exp equivalent of each type of list expression will now be displayed.

3.3.4.1. Explicit Lists

Explicit lists are easily desugared to simple expressions using the constructors: [ ] and PAIR. To understand how arbitrary explicit lists are transformed, it is enough to see how the following sample expressions are tranformed:

[1,2,3,4] becomes PAIR 1 (PAIR 2 (PAIR 3 (PAIR 4 [ ]))),

[flat,2,TIRE,1•23] becomes PAIR FLAT (PAIR 2 (PAIR TIRE 23)),[16]

[a•b] becomes PAIR A B, and

[A,b•(pair c [ ])] becomes PAIR A (PAIR B (PAIR C [ ])).

3.3.4.2. Arithmetic Sequence Expressions

Arithmetic sequence expressions are a convenient shorthand for monotonic sequences of numbers, where the k^{th} element (e_k) in the sequence may be expressed by: $e_1+(k-1)c$, for some constant c — i.e. arithmetic sequences. These sequences may be finite or infinite.

Finite arithmetic sequence exps take either the form [**X**,..,**Z**] or [**X**,**Y**,..,**Z**]; both of which are sugar for unknowns of the form:

FBT **X W Z**,

representing the sequence:

From **X** By **W** To **Z**,

where **W** is either 1 or (- **Y X**), respectively. Some finite arithmetic sequence exps and their SIMPLE-exp equivalents follow:

[16] The LNF system is case insensitive

[2,4,..,100] becomes FBT 2 (- 4 2) 100,

[1,..,1000] becomes FBT 1 1 1000, and

[10,7.5,..,0] becomes FBT 10 (- 7.5 10) 0.

Note that in a list of the form [**X**,..,**Y**] (without a second element), the second element is assumed to be **X**+1.

A sample (linearized) reduction of the finite arithmetic sequence exp: [2,4,..,100] to lazy-normal form:[17]

FBT 2 (- 4 2) 100 → FBT 2 2 100 → PAIR 2 (FBT′ 4 2 100).[18]

Infinite arithmetic sequence exps look like [**X**,..] or [**X**,**Y**,..] — both of which are transformed by the compiler to wffs taking the form:

FB **X** **W**,

representing the sequence:

From **X** By **W**,

where **W** is either 1 or (- **Y** **X**), respectively. Some sample transformations of infinite arithmetic sequence exps are displayed below:

[1,..] becomes FB 1 1,

[10,10,..] becomes FB 10 (- 10 10),

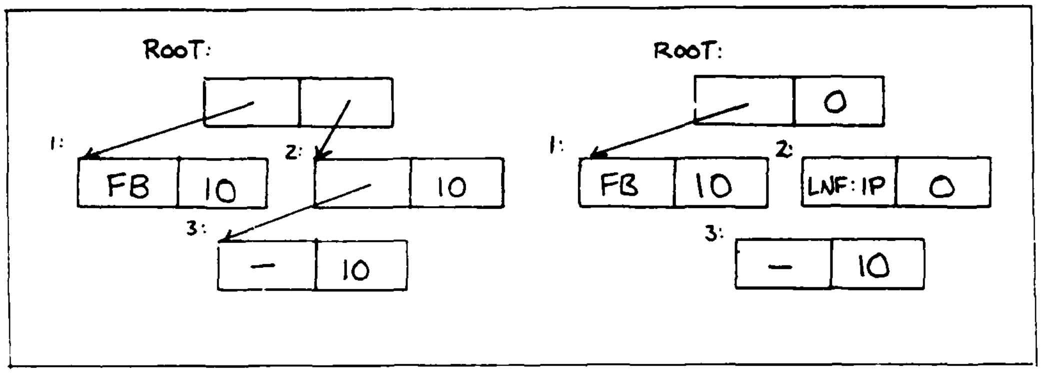
[1,3,..] becomes FB 1 (- 3 1), and

[0,-1,..] becomes FB 0 (- -1 0).

A graphical representation of the reduction of the sequence: [10,10,..] to lazy-normal form follows:

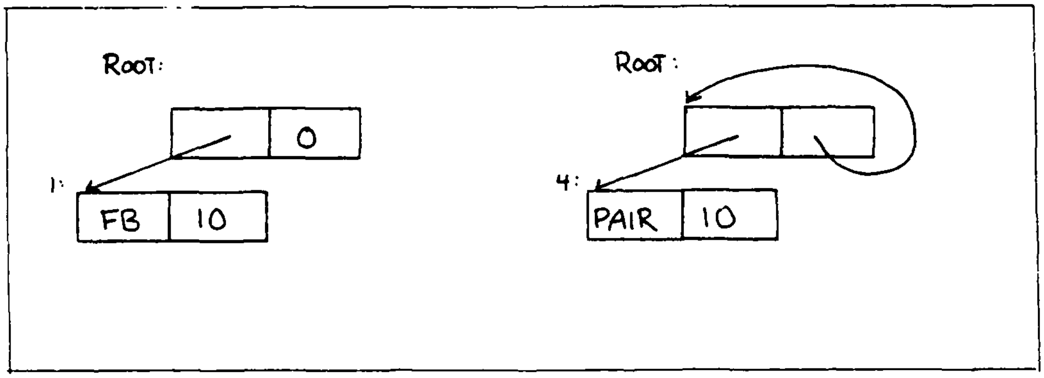
[17] The reader may, at this time, want to refer back to Chapter 2 for FBT's reduction rules

[18] FBT′ acts just like FBT except that it assumes its arguments have already been reduced to numbers.



FB 10 (- 10 10) reduces to FB 10 0
Figure 3.5

Recall that a CONS cell having the ZetaLisp atom LNF:IP as its CAR is the system's representation of a forwarding vertex.



FB 10 0 reduces to PAIR 10 (PAIR 10 (...))
Figure 3.6

Here is the second use made of cyclic graphs by LNF. The first use, as you may recall, was made by the fixed point finding functor Y.

3.3.4.3. Implicit Lists

The simple implicit list:[19]

```
for-each ?x∈[1,..,100]
  instantiate (sub1 (× 10 ?x))
```

reduces to the same list of numbers as does the arithmetic sequence expression: [9,19,..,999]. It is not, however, sugar for the same SIMPLE-exp as the arithmetic sequence. The implicit list above is sugar for the expression:

MAP (B SUB1 (× 10)) (FBT 1 1 100),

where the expression:

[19] The "for-each" implicit list syntax will be used exclusively in this section

B SUB1 (× 10)

is the result of compiling the LAMBDA-exp

λ (?x) (sub1 (× 10 ?x)).

To see that this compiled wff has the expected lazy-normal form — that is: PAIR **X** **REST**, where **X** is a wff which reduces to 9 and **REST** is a wff which reduces to the rest of the list — follow its two step reduction to lazy-normal form:

MAP (B SUB1 (× 10)) (FBT 1 1 100) →
MAP (B SUB1 (× 10)) (PAIR 1 (FBT′ 2 1 100)) →
PAIR (B SUB1 (× 10) 1) (MAP (B SUB1 (× 10)) (FBT′ 2 1 100)).

It should be (fairly) clear that the first argument to PAIR (the head of the list) reduces to 9. It should also be easy to see that the second argument, since it is just like the original LNF-wff except that (FBT 1 1 100) has been replaced with (FBT′ 2 1 100), will reduce to [19,...,999].

In general, an implicit list having the form:

for-each **be**∈**X**
  instantiate **BODY**

compiles to a SIMPLE-exp having the form:

MAP **FN** **LIST**,

where **FN** is the result of compiling the LAMBDA-exp:

(λ (**be**) **BODY**))

and **LIST** is the compiled version of **X**.

As illustrated by the two other examples of implicit lists above (see page 98), implicit lists may, in general, have a more complex structure than that just described. Besides always beginning with a phrase of the form: for-each **be**∈**X** (called a generator by Turner), and always ending with a phrase of the form: instantiate **BODY**, an implicit list may have one or more intervening phrases either having the form:

and-each **be**∈**X** (more generators)

or:

such-that **X** (called guards or filters).[20]

The FP language ALFL ([Hudak 1984c]) contains a similar, although restricted, construct called an "ordered bag". The first restriction is that all generators must precede all filters. More serious, although infinite lists are supported in the language, the ordered bag: [* [x,y] | x<-Nats; y<-Nats *] produces the list: [[1,1],[1,2],[1,3],[1,4],...] — a list in which most of the elements in the cross-product do not even appear!

To illustrate the scoping of an implicit list, consider the following for-each expression:

[20] Appendix B contains a BNF-like description of the syntax of implicit list expressions

```
for-each be₁∈LIST₁
  such-that GUARD₁
  and-for-each be₂∈LIST₂
    such-that GUARD₂
    instantiate EXP
```

The expressions in the scope of **be**$_1$'s variables are: **GUARD**$_1$, **LIST**$_2$, **GUARD**$_2$, and **EXP** — i.e. the expressions following the introduction of the bound expression **be**$_1$. Similarly, the expressions in the scope of the variables in **be**$_2$ are **GUARD**$_2$ and **EXP**. The expression **EXP** is called the template of the implicit list.

An expression having the above form is transformed into an equivalent combination (see below), and then compiled.

```
ENUMERATE
 (MAP (λ (be₁)
       (IF GUARD₁
          (MAP (λ (be₂) EXP)
             (FILTER (λ (be₂) GUARD₂)
                    LIST₂))
          [ ]))
     LIST₁)
```

Careful inspection of this rather complicated expression reveals that it reduces to the expected construction — a (possibly empty) list of instantiated **EXP**s. To understand the expression, one must be familiar with the workings of the functors: MAP, FILTER, and ENUMERATE. The rules defining the functors MAP and FILTER are straightforward (see Chapter 2), but the rules which define ENUMERATE are not. ENUMERATE may best be understood not by peering at its rule and the rules of the other functors upon which its rule depends (TURN, UP, and DOWN), but by seeing what kind of construction it expects as an argument and what kind of construction it produces from that argument.

ENUMERATE expects as argument a list (empty, finite, or infinite) of lists, each of which may also be empty, finite, or infinite. That is to say, an appropriate argument for ENUMERATE takes the form:

$$
\begin{array}{l}
[[\mathbf{X}_{11},\mathbf{X}_{12},\mathbf{X}_{13},\ldots],\\
\;[\mathbf{X}_{21},\mathbf{X}_{22},\mathbf{X}_{23},\ldots],\\
\;[\mathbf{X}_{31},\mathbf{X}_{32},\mathbf{X}_{33},\ldots],\\
\;\ldots].
\end{array}
$$

ENUMERATE, applied to such a list, reduces to the list:

$$[\mathbf{X}_{11},\mathbf{X}_{12},\mathbf{X}_{21},\mathbf{X}_{31},\mathbf{X}_{22},\mathbf{X}_{13},\mathbf{X}_{14},\mathbf{X}_{23},\ldots].$$

Thus ENUMERATE borrows the scheme Cantor used for demonstrating the countability of the rationals and produces a flattened list containing all of the elements in each of its argument's sublists. The rules defining ENUMERATE and its "helping" functors were gleaned from a functional definition of ENUMERATE by F. L. Morris, [Morris 1984].

Turner, for his ZF expressions in KRC, uses a different implementation strategy involving the functors FLATMAP and INTERLEAVE — instead of ENUMERATE and MAP.[21] The main difference between this contruct's implementation in LNF and KRC is the order in which the elements of the implicit list are produced. Turner's implementation is biased more towards the first generator — i.e. the first list in a ZF expression is "run through" much more quickly than the rest of the lists.

An implicit list, viewed as an initial phrase **P** (which may be either a generating or filtering phrase) and remaining phrases **R**, is *transform*ed as follows. In case:

P is **be**∈**X** and **R** consists of just a template:
MAP (λ (**be**) **R**) **X**

P is **be**∈**X** and **R** contains only guards **GS** and a template **T**:
MAP (λ (**be**) **T**)
(FILTER (λ (**be**) (conjunction of the **GS**)) **X**)

P is **be**∈**X** and **R** contains generators:
ENUMERATE (MAP (λ (**be**) (*transform* **R**)) **X**)

P is a guard:
IF **P** (*transform* **R**) [ ]

The implicit list:

```
for-each ?x∈[1,..,5]
    such-that (odd ?x)
    and-for-each ?y∈[100,101]
    instantiate [?x•?y]
```

is *transform*ed to the combination:

```
ENUMERATE
  (MAP
     (C (S' IF ODD (C' MAP PAIR (PAIR 100 (PAIR 101 [ ])))) [ ])
     (FBT 1 1 5)).
```

This compiled implicit list reduces to its lazy-normal form:

```
[[1•100]•
  UP [ ]
     [MAP (PAIR 1) [101]]
     (MAP (C (S' IF ODD (C' MAP PAIR [100,101])) [ ]) (FBT' 2 1 5))]
```

in 14 reduction steps.

3.3.5. Conditional Expressions

There are two conditional expressions in the LNF language: IF expressions and CASE expressions.

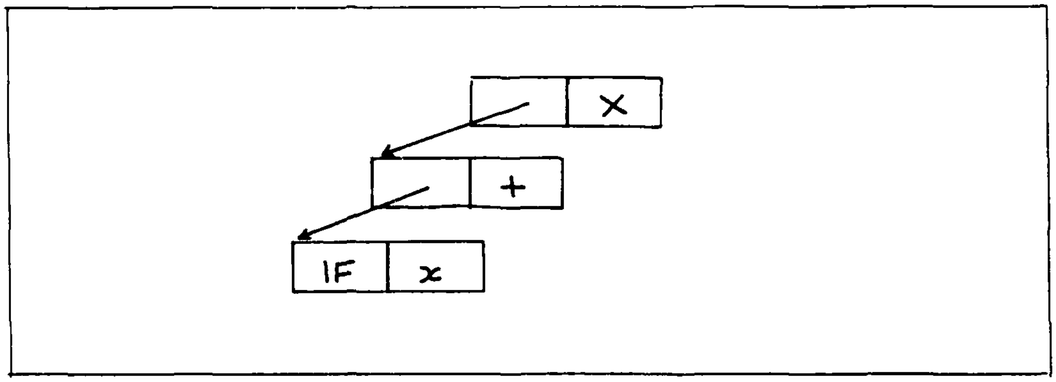
[21] Turner's implementation scheme is explained quite nicely in [Abelson 1985]

3.3.5.1. IF Expressions

The IF-exp:

if **CONDITION** then **THEN-EXP** else **ELSE-EXP**

is simply sugar for a combination having operator: (IF **CONDITION THEN-EXP**) and operand: **ELSE-EXP**. Its representation, therefore, takes the same form as any combination having three arguments.



ZetaLisp representation of the conditional: if x then + else ×

Figure 3.7

3.3.5.2. CASE Expressions

CASE expressions (CASE-exps), introduced in the discussion of function declarations, are conditional binding constructs.[22] The CASE-exp:

case **E** in
  $\mathbf{cbe}_1 \rightarrow \mathbf{BODY}_1$ |
  $\mathbf{cbe}_2 \rightarrow \mathbf{BODY}_2$ |
  ...
  $\mathbf{cbe}_n \rightarrow \mathbf{BODY}_n$
endcase

attempts to match the object of the case (E) against the pairwise non-unifiable ([Robinson 1965]) case templates ($\mathbf{cbe}_i$ s) — which are just constructed bound expressions. If E matches template $\mathbf{cbe}_j$, then the case expression reduces to (the compiled equivalent of the β-redex):

$(\lambda\ (\mathbf{cbe}_j)\ \mathbf{BODY}_j)\ \mathrm{E}.$

If E does not match any of the templates, then the CASE-exp reduces to an unknown. A CASE-exp is transformed to a combination, employing the functors A-S-E, A-S-E$'$, and A-S$'$. The A-S$'$ functor is a nonstrict version of the A-S functor — inasmuch as it does not reduce its fourth argument. The functors A-S-E and A-S-E$'$ are best explained by studying A-S-E's four reduction rules, which are:

[22] Other FP languages which contain similar constructs include ML ([Milner 1983]), Lazy ML ([Augustsson 1984a], [Augustsson 1984b], [Johnsson 1981b], [Johnsson 1983], and [Johnsson 1984]), and HASL ([Abramson 1982b] and [Abramson 1983])

A-S-E A-S-E $\mathbf{c}\ \mathbf{i}\ \mathbf{X}\ \mathbf{Y}\ (\mathbf{c}\ \ \mathbf{Z}_1 \cdots \mathbf{Z}_i) \rightarrow \mathbf{X}$
A-S-E $\mathbf{c}_1\ \mathbf{i}\ \mathbf{X}\ \mathbf{Y}\ (\mathbf{c}_2\ \mathbf{Z}_1 \cdots \mathbf{Z}_j) \rightarrow \mathbf{Y}$,
if $\underline{\mathbf{c}_1 \neq \mathbf{c}_2 \text{ or } \mathbf{i} \neq \mathbf{j}}$
A-S-E $\mathbf{c}\ \mathbf{i}\ \mathbf{X}\ \mathbf{Y}$ **FN** → **Y**
A-S-E $\mathbf{c}\ \mathbf{i}\ \mathbf{X}\ \mathbf{Y}$ **RDU** → A-S-E $\mathbf{c}\ \mathbf{i}\ \mathbf{X}\ \mathbf{Y}$ **IMR**

Together, these rules mean that the LNF-wff:

A-S-E **c i X Y Z**

is reduced just like the wff:

```
IF (AND (= c (CONSTRUCTOR Z)) (= i (NUM-ARGS Z)))
  X
  Y
```

A-S-E is a condensed form of "Abstract Structure Else". The functor A-S-E′ is to A-S-E as A-S′ is to A-S. CASE-exps are compiled by the function Compile-case and its three helping functions: Abstract-cases, Abstract-template-else, and Abstract-templates-else which appear below:

```
(DEFUN Compile-case (case-exp)
 (LET ((cases (Cases case-exp))
       (case-object (Case-object case-exp))
       (var (New-variable)))
   (IF (Order-dependent cases) (Issue-warning-message))
   (Combine (C-T-abs var (Abstract-cases cases var)) case-object)))

(DEFUN Abstract-cases
 (cases var &optional (already-seen-a-case NIL))
 (LET* ((first-case (CAR cases))
       (rest-cases (CDR cases))
       (template (Template first-case))
       (result (Result first-case)))
   (IF (NULL rest-cases) ;; FIRST CASE IS ALSO THE LAST CASE
       (Combine (Abstract-be template result already-seen-a-case) var)
       (Abstract-template-else
         template
         result
         var
         (Abstract-cases rest-cases var T)
         already-seen-a-case))))
```

```
(DEFUN Abstract-template-else
 (template result var else &optional (already-seen-a-case NIL))
 (Combine
   (IF (Constructed-be-p template)
       (LET ((constructor (Constructor template))
             (num-args (Num-args template)))
        (A-S-E-or-A-S-E'-comb
          already-seen-a-case
          constructor
          num-args
          (Abstract-templates-else
            (Args template)
            result
            var
            1
            else)
          else)
       ;; TEMPLATE IS A VARIABLE, SO NO NEED FOR ELSE
       (C-T-abs template result))
   var))

(DEFUN Abstract-templates-else
 (templates result var arg-number else)
 (IF (NULL templates)
     result
     (Abstract-template-else
       (CAR templates)
       (Abstract-templates-else
         (CDR templates)
         result
         var
         (ADD1 arg-number)
         else)
       (Combine (Combine 'ARG arg-number) var)
       else)))
```

Note that if the piece of code:

```
(Combine (C-T-abs var (Abstract-cases cases var)) case-object))
```

in Compile-case was replaced with:

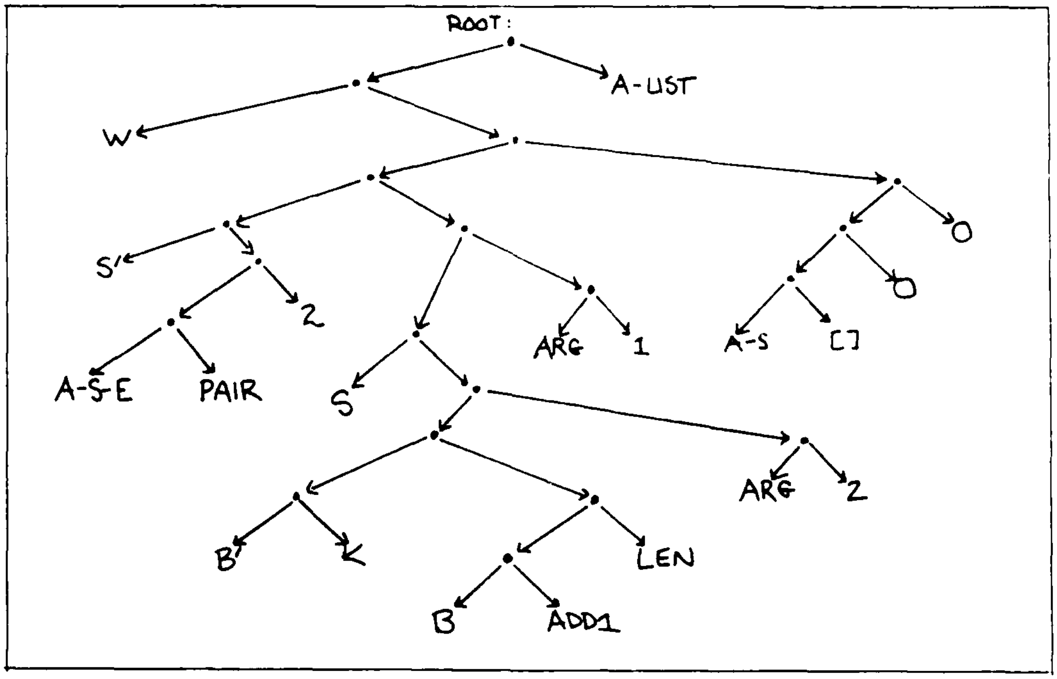
```
(Abstract-cases cases case-object)
```

then CASE-exps would not be fully lazy. In situations where case-object is an unknown containing variables — e.g. (+ 1 ?x) — more than one redex may be created and reduced, violating the property of full-laziness LNF enjoys.

Two concrete CASE-exps and their compiled equivalents follow. The CASE-exp:

```
case a-list in
  [?•?r] → add1 (len ?r) |
  [ ] → 0
endcase
```

compiles to (the LNF-wff):

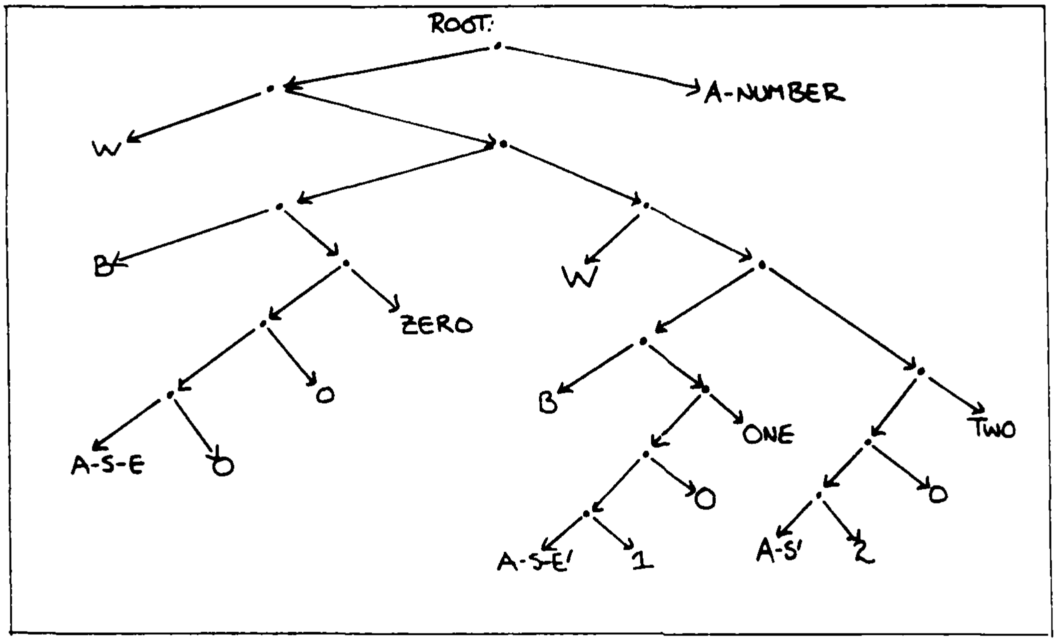


Compiled code of the CASE-exp above
Figure 3.8

The CASE-exp:

```
case a-number in
  0 → zero |
  1 → one  |
  2 → two
endcase
```

compiles to (the LNF-wff):



Compiled code of the CASE-exp above
Figure 3.9

3.3.6. Compiler Summary

It has been shown how each type of LNF expression may be transformed into a simple LNF expression and how simple LNF expressions may be transformed into representations of LNF-wffs. In essence, the transformation's task (except for some minor bits of desugaring) is the elimination of bound variables in favor of LNF-wffs. The next section will detail the mechanisms which transform these LNF-wff representations into lazy-normal form.

3.4. LNF's Runtime Environment

Since a compiled LNF program is not a fixed sequence of instructions to a Von Neumann style machine but is a representation of an LNF-wff — i.e. a graph in which program and data are indistinguishable; running such a program will involve manipulating LNF-wffs.

LNF's runtime system (implemented by the routine LNF-of-wff and its subsidiaries) is a realization of the machine called LNF-M in Chapter 2. Recall that LNF-M, given an LNF-wff **X** as input, either terminates, yielding an LNF-wff **LNFX** such that **X** LNF-red* **LNFX** and **LNFX** in lazy-normal form, or does not terminate, in which case **X** has no lazy-normal form.

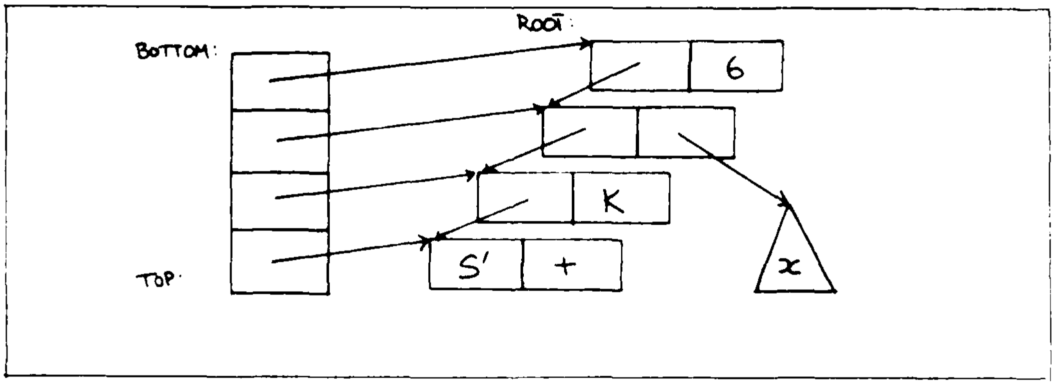
LNF-of-wff has a simple yet flexible organization — it is composed of two collections of routines. One collection is responsible for controlling the reduction of an LNF-wff to lazy-normal form and the other collection is responsible for performing the individual reduction steps. The routines which control the reduction are independent of the system's set of functors — they could also be used (as is) in a realization of the SKI-G-calculus. The routines which perform the individual reduction steps are mutually independent and functor specific — there is one routine per functor. The functor specific routine for functor **f** (called **f**-reduce) is, in essence, an encoding of **f**'s LNF-calculus reduction rules and is responsible for reducing (if reducible) a wff having **f** as its initial atom. This organization facilitates experimentation with different functor sets, as functors may be added to (removed from) the system by simply adding (removing) functor specific routines — no code need be modified.

Although far from being a specification for a piece of hardware, the implementation is quite machine-like. That is to say the routines themselves are written in an imperative and "referentially opaque" style. The machine-like structure of the runtime system's implementation was determined in part by a plan to move the implementation (or some successor of it) out of software and into firmware and maybe even to hardware.

All of the significant routines making up LNF's runtime system and the data structures which they employ will now be discussed in detail. The routines which control the reduction (which are the top level routines in the runtime system) are discussed first.

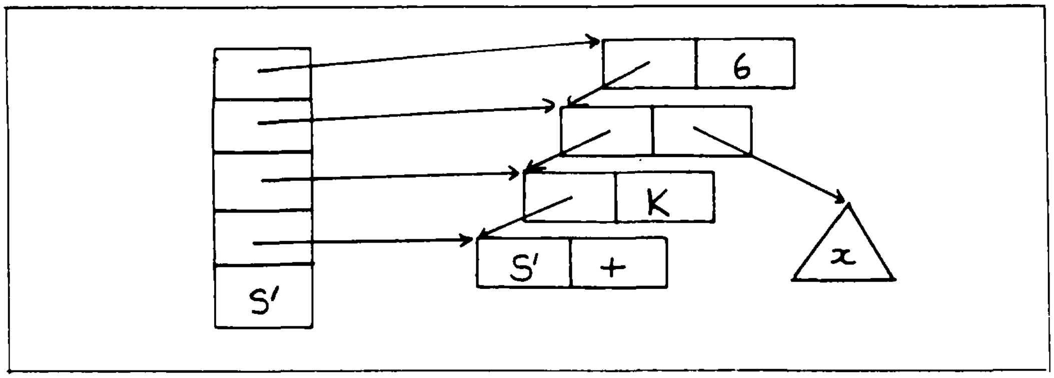
3.4.1. Controlling the Reduction

The routines controlling the reduction of an LNF-wff employ a stack; the items in the stack are stacks (called left ancestor stacks) themselves. A *left ancestor stack* (LAS) is the key data structure used in D.A. Turner's implementation of SASL — outlined in [Turner 1979c]. An LAS, used in conjunction with an expression graph (an LNF-wff), eases access to the LNF-wff's initial atom and arguments. The bottom item of such a stack points at the root of the LNF-wff. Each of the stack's other items points at the operator of the LNF-wff pointed at by the item just below it. An LAS representation is called *canonical* if its top item is the LNF-wff's initial atom.



An Example of a (Non-canonical) Left Ancestor Stack
Figure 3.10

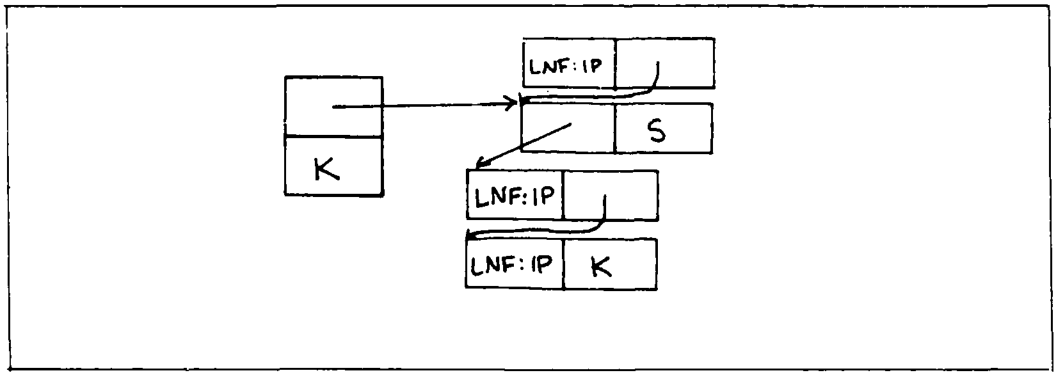
It is convenient to display the LASs growing downward, since the trees (LNF-wffs) they represent are customarily pictured with root at top and leaves at the bottom.



An Example of a Canonical Left Ancestor Stack
Figure 3.11

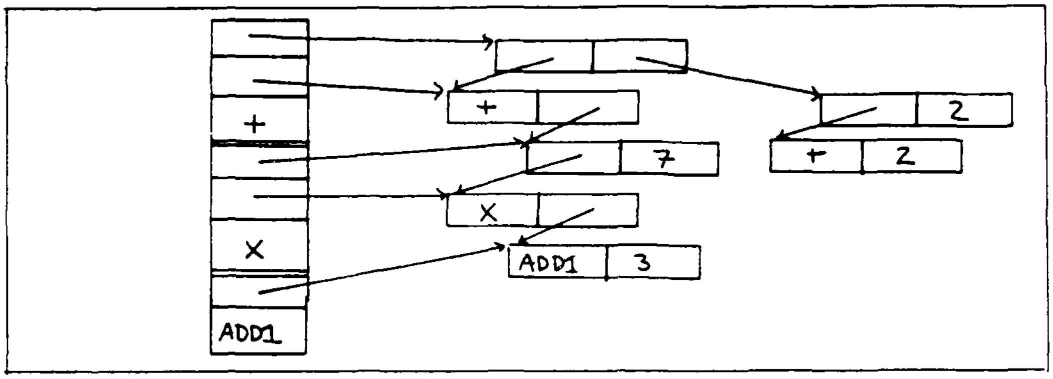
One can see that a canonical LAS facilitates access to the LNF-wff's initial atom and arguments. If an LAS is realized by an array (as is done in this implementation) the LNF-wff's initial atom and arguments may be accessed in constant time.

The next example illustrates the other property of LASs — no canonical LAS item points to a forwarding vertex. The top item of a non-canonical LAS may point to a forwarding vertex. It will be seen that the functor specific routines access the LNF-wff's arguments via a canonical LAS. The fact that the LAS's items are never forwarding vertices ensures that these routines will have to handle only "real" LNF-wffs — i.e. combinations and atoms.



Stack Items Do Not Point at Forwarding Vertices
Figure 3.12

It was stated above that the runtime system employed a stack of LASs. Briefly, the stack of LASs is used to locate the next redex to be reduced. The bottom item is the LAS representing the whole LNF-wff. If this LNF-wff is a reduction context for argument i, then the next item will be the LAS representing the LNF-wff's ith argument The top LAS represents the LNF-wff on which the system is currently focusing its attention. An example follows:



A Stack of LASs representing: + (× (add1 3) 7) (+ 2 2)
Figure 3.13

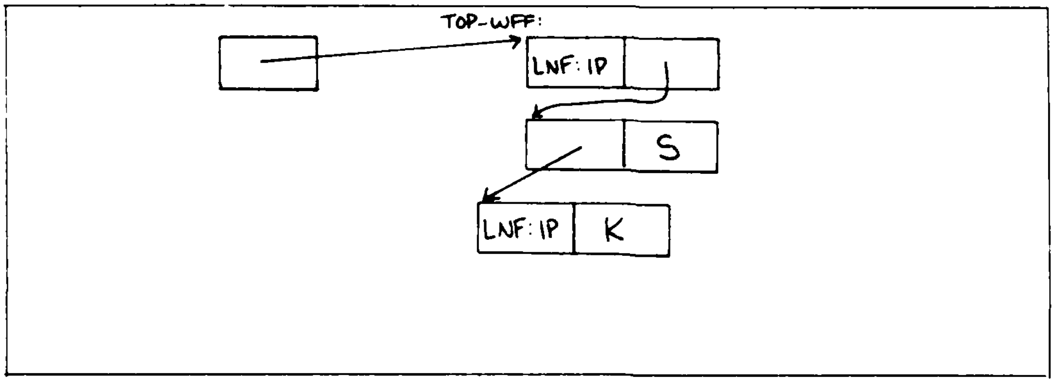
In the discussions to follow, the stack of LASs will be referred to as simply the stack; its items (which are also stacks) will be referred to as the LASs. The ZetaLisp code which realizes this system, starting with the code for LNF-of-wff, will now be presented.

```
;; Returns as value the lazy-normal form of wff (if one exists).
;; Assumes nothing about current state of the stack.
(DEFUN LNF-of-wff (wff)
  ;; First, clear the stack.
  (Clear-stack)
  ;; Then, reduce the wff to lazy-normal form and return it.
  (LNF-of-subwff wff))

;; Returns the lazy-normal form of wff, leaves the stack unchanged.
(DEFUN LNF-of-subwff (wff)
  ;; Find the wff's lazy normal form,
  ;; leaving its LAS representation as the stack's top element.
  (Stack-of-LNF-of-subwff wff)
  ;; Pop the top (canonical) LAS off of the stack,
  ;; then return that LAS's bottom element as result.
  (Pop-stack))

;; Reduces wff to lazy-normal form and
;; places its canonical LAS representation on top of stack.
;; It is called for these side effects only.
(DEFUN Stack-of-LNF-of-subwff (wff)
  ;; Push (non-canonical) LAS representation of wff on stack.
  (Push-stack wff)
  ;; Reduce wff represented in top LAS to lazy-normal form.
  ;; Leave canonical LAS representation of it on top.
  (Reduce-stack-to-LNF))
```

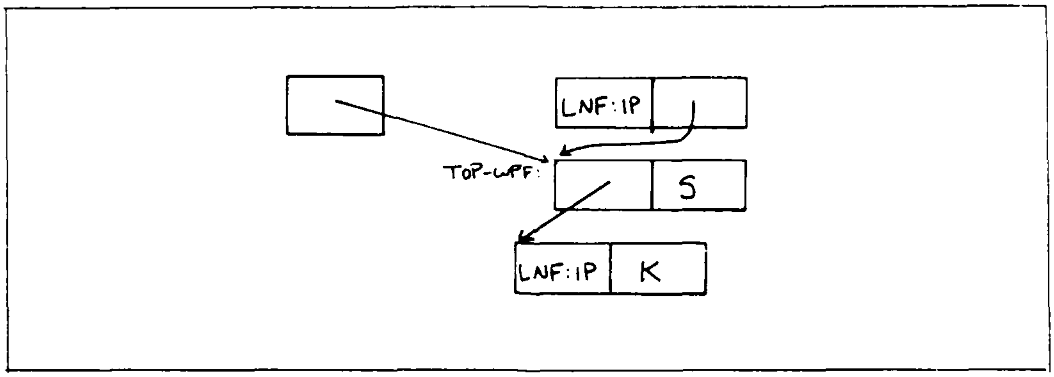
The following function is "the execution cycle" of the runtime system.

```
;; Assumes a non-canonical LAS on top of stack.
;; Reduces the LNF-wff it's representing to lazy-normal form,
;; leaving the canonical LAS of this reduced LNF-wff on top.
;; Called for these side effects only.
(DEFUN Reduce-stack-to-LNF ()
  (LOOP ;; is exited when (RETURN) is evaluated.
    ;; Canonicalize top LAS on stack.
    (Canonicalize-stack)
    ;; Attempt to reduce initial redex.
    ;; This may involve reducing some arguments first.
    ;; If no initial reduction performed or reduction makes
    ;; LNF-wff irreducible, then return.
    (LET ((reduction-code (Attempt-initial-reduction)))
      (IF (OR (Reduction-not-performed reduction-code)
              (LNF-wff-now-irreducible reduction-code))
          (RETURN)))))

;; Assumes stack is not empty. Canonicalizes the top
;; LAS. Called for its side effect on the LAS only.
(DEFUN Canonicalize-stack ()
  (LET ((top-wff (Top-wff-on-top-LAS)))
    (LOOP WHILE (NOT (Atom-p top-wff))
      ;; top-wff is either a combination or
      ;; a forwarding vertex
      (IF (Combination-p top-wff)
        ;; Assign top-wff to be its own operator
        ;; Push top-wff onto the top of the LAS
        (Push-top-LAS (SETQ top-wff (Operator top-wff)))
        ;; Otherwise, top-wff is a forwarding vertex, so
        ;; Assign top-wff to be the LNF-wff to which it was
        ;; forwarded. Overwrite LAS's top item with new top-wff.
        (Replace-LAS-top (SETQ top-wff (Forwarded-to top-wff)))))))
```

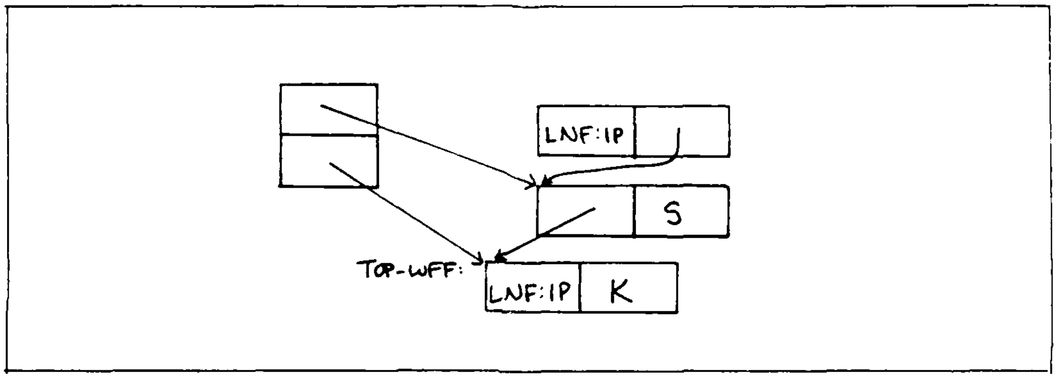
A step by step example of LAS canonicalization follows.



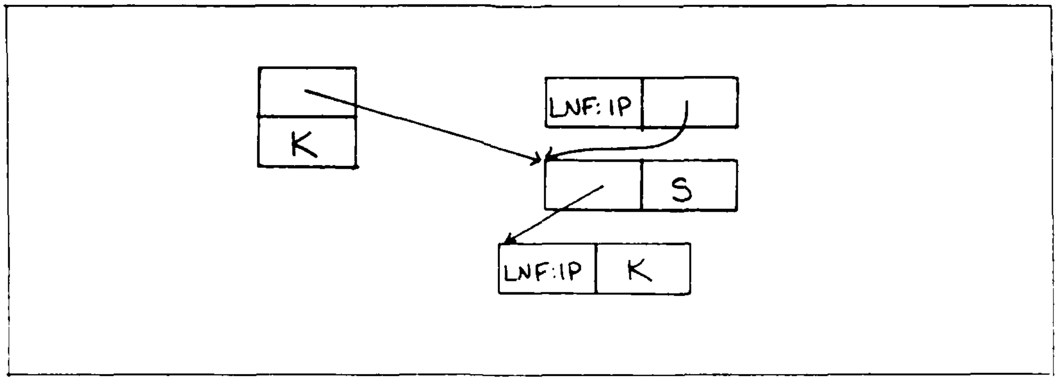
Just After Initialization
Figure 3.14



After One Trip Through the Loop
Figure 3.15



After Two Trips Through the Loop
Figure 3.16



Top LAS has been Canonicalized
Figure 3.17

Observe that there is no "loop check" in the routine Canonicalize-stack. Thus, an LNF-wff having a cyclic "left spine" will cause the system to run forever. The decision to leave the check out was made because such LNF-wffs have no lazy-normal form anyway, and the system does not claim to terminate for LNF-wffs having no lazy-normal form.

The one remaining control routine to be displayed[23] is Attempt-initial-reduction. Its job

[23] The very low level routines like Replace-LAS-top, Push-stack, Push-top-LAS, will not be displayed

is to try to perform *a single initial reduction on the LNF-wff* (represented by the top LAS). To do this it is sometimes necessary to perform some internal reductions first. These internal reductions are performed if and only if the LNF-wff's initial atom is a strict (or partially strict — strict for just some of its arguments) functor.

```
;; Assumes canonical LAS on top of stack
;; Returns a code informing caller whether or not an initial
;; reduction was performed. If one was performed, the code
;; informs the caller whether or not the reduction has made the
;; LNF-wff irreducible.
(DEFUN Attempt-initial-reduction ()
 (LET* ((initial-atom (Top-of-top-LAS))
      ;; Initial-atom is a constructor or a functor.
      ;; It is a functor iff there is a reduction routine
      ;; for it on its property list.
      (functor-specific-reduction-routine
        (GET initial-atom 'LNF:REDUCER)))
      ;; functor-specific-reduction-routine is either NIL,
      ;; in which case initial-atom is a constructor, or
      ;; it is the routine responsible for reducing
      ;; LNF-wffs having initial-atom as their initial atom.
   (IF (AND functor-specific-reduction-routine
          ;; initial-atom is a functor
          (≤ (GET initial-atom 'LNF:ARITY)
             (Number-args-in-top-LAS))
          ;; there are enough arguments to form a redex
      ;; then run the routine!
      ;; It will return a reduction code.
      (FUNCALL functor-specific-reduction-routine)
      ;; Otherwise, LNF-wff is a construction or
      ;; function so its already in lazy-normal form.
      ;; Return "no reduction performed" code.
      *NO-RED*)))
```

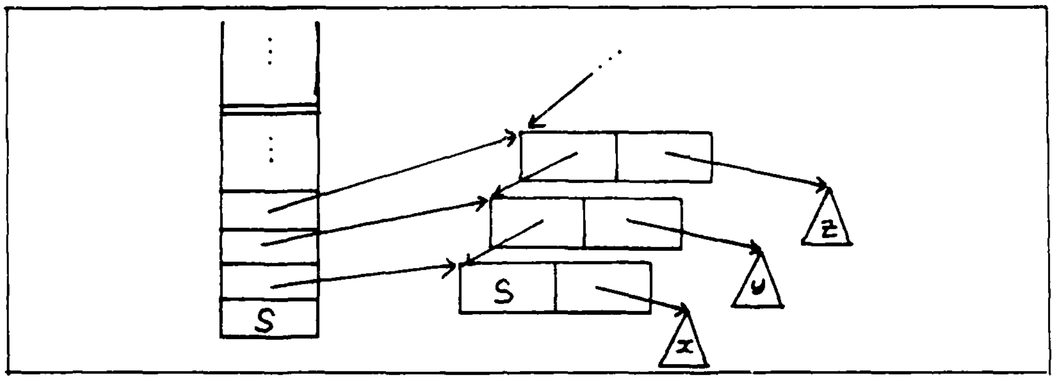
This completes the discussion of the runtime system's control routines. The next section details several of the functor specific reduction routines. Also presented in the next section will be a detailed example illustrating the workings of the system.

3.4.2. The Functor Specific Reduction Routines

As stated above, there is one reduction routine for each functor. The reduction routine for functor **f** (**f**-reduce) expects the top item of the stack to be a canonical LAS repesentation of an LNF-wff of the form:

$$\mathbf{f}\,\mathbf{X}_1 \cdots \mathbf{X}_n\text{, where } n \geq \text{ARITY}[\mathbf{f}]$$

For example, the routine S-reduce expects the stack to look like (recall the functor S has arity 3):

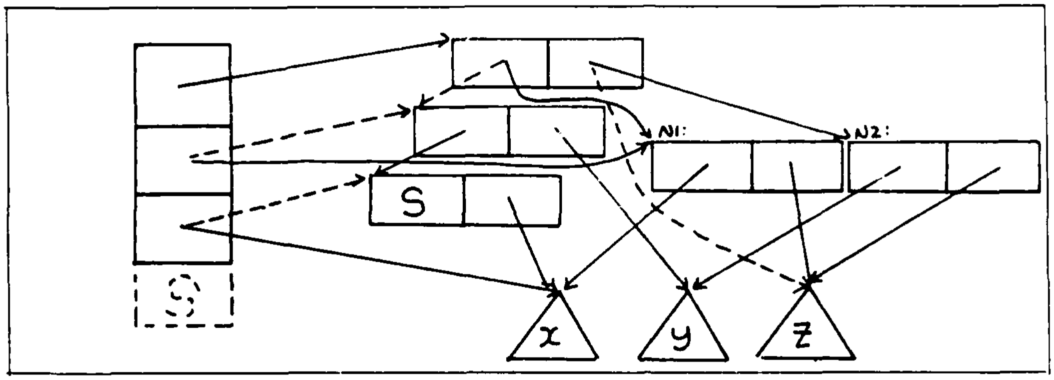


Possible State of System When S-reduce Begins
Figure 3.18

The code for the S-reduce routine follows.

```
;; S X Y Z → X Z (Y Z)
(DEFUN S-reduce ()
  (LET* ((redex (LAS-item-4))
         (x (LAS-arg-1))
         (y (LAS-arg-2))
         (z (LAS-arg-3))
         ;; create the new combination X Z
         (xz (Combine x z))
         ;; create the new combination Y Z
         (yz (Combine y z)))
    ;; Overwrite the operator and operand of redex with
    ;; xz and yz respectively.
    (Replace-operator-and-operand redex xz xy)
    ;; Overwrite item which used to contain Sxy with xz.
    (Replace-LAS-item-3 xz)
    ;; Overwrite item which used to contain Sx with x.
    (Replace-LAS-item-2 x)
    ;; Pop the functor S from LAS.
    (Pop-LAS)
    ;; Return the S reduction code.
    *RTP-S*))

;; Overwrites comb's operator and operand with newopr and
;; newopd, respectively. Called for its side effect only.
(DEFUN Replace-operator-and-operand (comb newopr newopd)
  (RPLACD (RPLACA comb newopr) newopd))
```

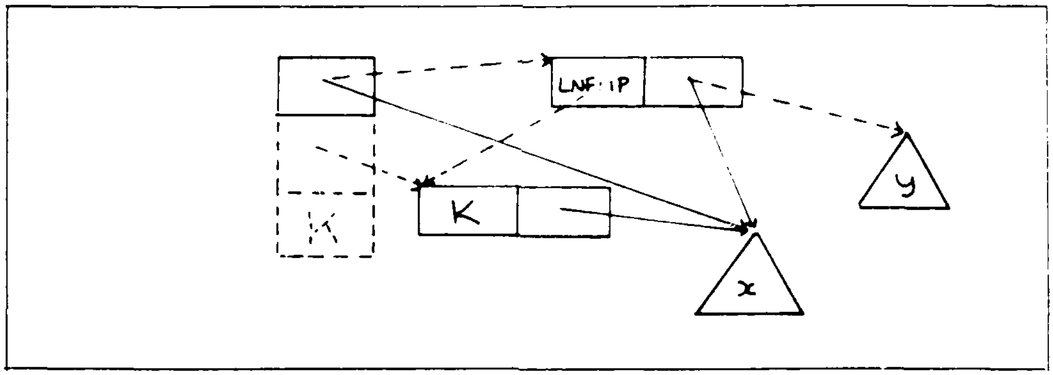
A minor point — in S-reduce, the two LAS stack overwrite operations and the popping of the LAS may be replaced with the simpler: (Pop-n-items-from-LAS 3) since the next call on Canonicalize-stack will perform these overwritings. The overwriting is performed in S-reduce just because the system knows it will have to be done soon and since it has the wffs in hand, why not do it? A graphical representation of S-reduce's operation follows.



The Workings of S-reduce
Figure 3.19

Note that two new combinations have been created by S-reduce. One may assign a "space cost" to each of the reduction routines — the number of combination cells created. The space cost of S-reduce is therefore equal to two. The code for the K-reduce routine follows.

```
;; K X Y → X
(DEFUN K-reduce ()
 (Forward-combination
   ;; the redex
   (LAS-item-3)
   ;; to X
   (LAS-arg-1)
   ;; then pop the LAS twice, then
   ;; replaces its top item with wff
   2)
 ;; return the K reduction code
 *RTP-K*)

;; Forwards comb to wff and pops top LAS n times.
;; Called for its side effects only.
(DEFUN Forward-combination (comb wff &optional n)
 (Replace-operator-and-operand comb 'LNF:IP wff)
 (COND (n ;; n is NIL if not provided as argument
        (Pop-n-items-from-LAS n)
        (Replace-LAS-top wff))))
```

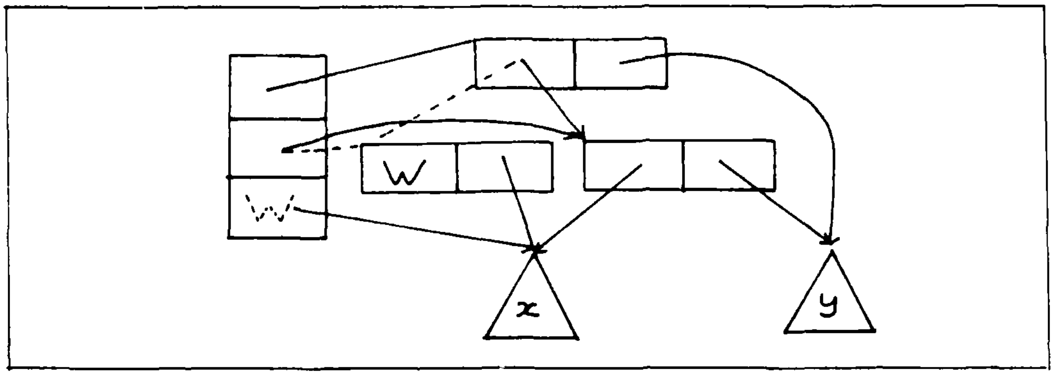


The Workings of K-reduce
Figure 3.20

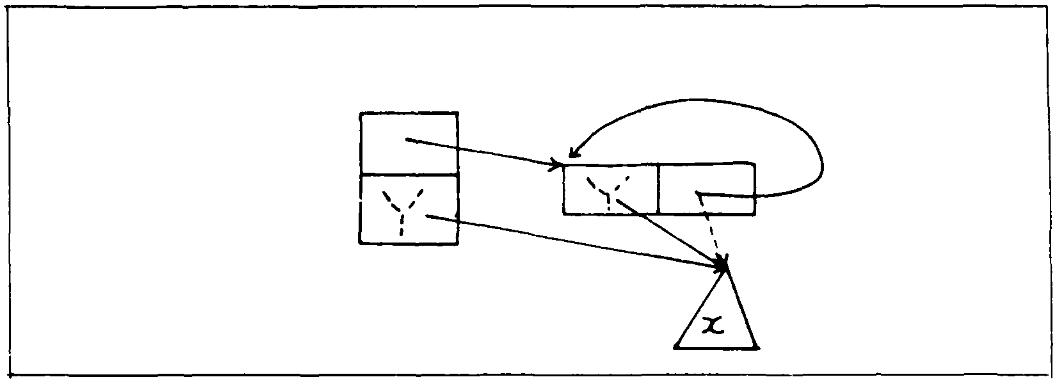
Observe that, following the K reduction, LAS's top item is **X** and not the forwarding vertex which points at **X**. This is another case of a reduction routine doing a job that Canonicalize-stack would have to do later. K-reduce's space cost is zero The W-reduce and Y-reduce routines are presented next.

```
;; W X Y → X Y Y
(DEFUN W-reduce ()
  (LET* ((redex (LAS-item-3))
         (x (LAS-arg-1))
         (y (LAS-arg-2))
         ;; Create the new combination X Y
         (xy (Combine x y)))
    ;; Overwrite redex's operator with xy
    (Replace-operator redex xy)
    ;; Overwrite item that used to contain Wx with xy
    (Replace-LAS-item-2 xy)
    ;; Overwrite item that used to contain W with x
    (Replace-LAS-top x)
    ;; Return W reduction code.
    *RTP-W*))

;; Y X → X (X (X ...))
(DEFUN Y-reduce ()
  (LET* ((redex (LAS-item-2))
         (x (LAS-arg-2)))
    ;; Overwrite redex's operator with x and operand with
    ;; itself!
    (Replace-operator-and-operand redex x redex)
    ;; Overwrite item which used to contain Y with x.
    (Replace-LAS-top x)
    ;; Return Y reduction code.
    *RTP-Y*))
```

The W-reduce routine costs one combinat on while the Y-reduce routine costs nothing at all to run.



The Workings of W-reduce
Figure 3.21

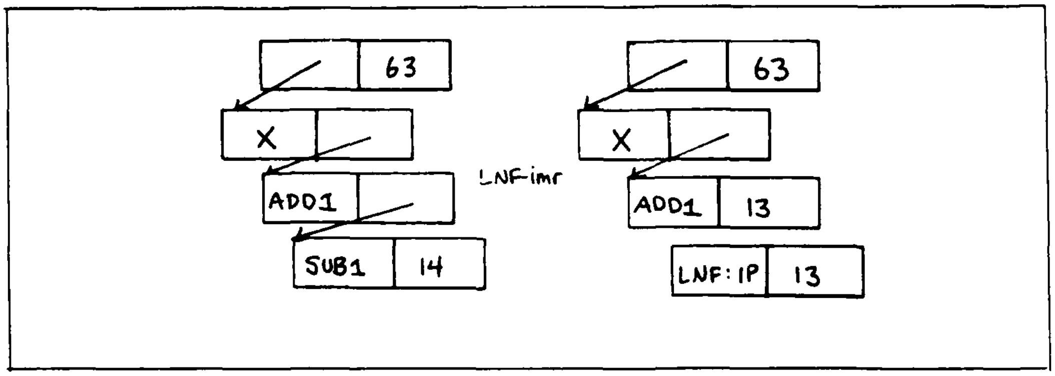


The Workings of Y-reduce
Figure 3.22

The rest of the routines specific to non-strict functors (B, C, S′, B′, ...) are implemented in a similar fashion. Some reduction routines which deal with strict functors will now be detailed. The first to be presented is ×-reduce.

```
;; × n m → n×m
;; × RDU Y → × IMR Y
;; × n RDU → × n IMR
(DEFUN ×-reduce ()
  (LET ((redex (LAS-item-3))
        (x (LNF-of-subwff (LAS-arg-1))))
    (IF (NUMBERP x) ;; THEN
        (LET ((y (LNF-of-subwff (LAS-arg-2))))
          (COND ((NUMBERP y)
                 (Forward-combination
                   redex ;; to
                   (× x y)
                   ;; Pop the LAS twice, then
                   ;; replaces its top item with (× x y).
                   2)
                ;; Return code which informs caller that
                ;; × reduction was performed and LNF-wff
                ;; now irreducible.
                (Irreducible-code *RTP-×*))
               (T ;; y's lazy-normal form is not a number, so
                ;; return "no reduction performed" code.
                *NO-RED*)))
        ;; ELSE x's lazy-normal form not a number, so
        ;; return "no reduction performed" code.
        *NO-RED*)))
```

The above routine requires some explanation. It purports to be an encoding of the three reduction rules for multiplication (the three comment lines just preceding the code). Where are these rules in the code? Before answering this question, there is an obvious but (pragmatically) important point to be made concerning reduction contexts in the LNF-calculus. If **X** is a reducible **f** reduction context for argument **i** and **X** LNF-imr **Y** (**Y** is just like **X** except that ARG[i,X] has been reduced to ARG[i,Y]) and ARG[i,Y] reducible, then **Y** is an **f** reduction context for argument **i**. For example, the LNF-wff on the left in the following figure is a × reduction context for argument one. The LNF-wff on the right (the reductum of the LNF-wff on the left) is also a × reduction context for argument one.



Example of Reduction Context Preservation
Figure 3.23

Because reduction contexts are preserved in this way ×-reduce may reduce its LNF-wff's first argument all the way to lazy-normal form (via LNF-of-subwff[24]), rather than just performing a single reduction on it (as its first contextual reduction rule specifies). After the first argument has been reduced, it is time to check and see if it reduced to a number. If it did, then the LNF-wff is now a reduction context for the second argument. The routine proceeds to reduce the second argument to lazy-normal form (again via LNF-of-subwff). If the second argument is a number, then ×'s substantive reduction rule may be applied.

Thus ×'s first contextual rule is endcoded in the routine's third line:

```
(x (LNF-of-subwff (LAS-arg-1))))
```

and ×'s second contextual rule is hidden in lines four and five:

```
(IF (NUMBERP x) ;; THEN
   (LET ((y (LNF-of-subwff (LAS-arg-2)))).
```

Its only substantive reduction rule is realized by the two nested predications (NUMBERP x) and (NUMBERP y) and the call on the function Forward-combination which forwards the redex to the product of x and y.

All of the routines which deal with strict functors follow a reduction sequence similar to that followed by ×-reduce. First the routine finds the appropriate argument to reduce (determined by the functor's contextual reduction rules). That argument is reduced to lazy-normal form. If the reduction of that argument creates a reduction context for another argument, then that argument is reduced. When all of the functor's contextual reduction rules have been applied, then the routine tries to apply a substantive reduction rule.

Enough reduction rules have been presented now to enable a not totally trivial example of LNF-wff reduction to be given. The LNF-wff to be reduced in this example is:

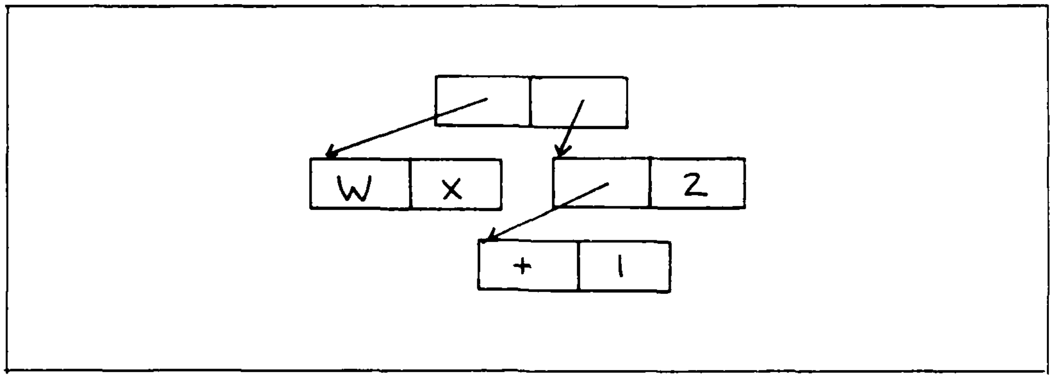
W × (+ 1 2)

which is the LNF-wff to which the LNF-exp

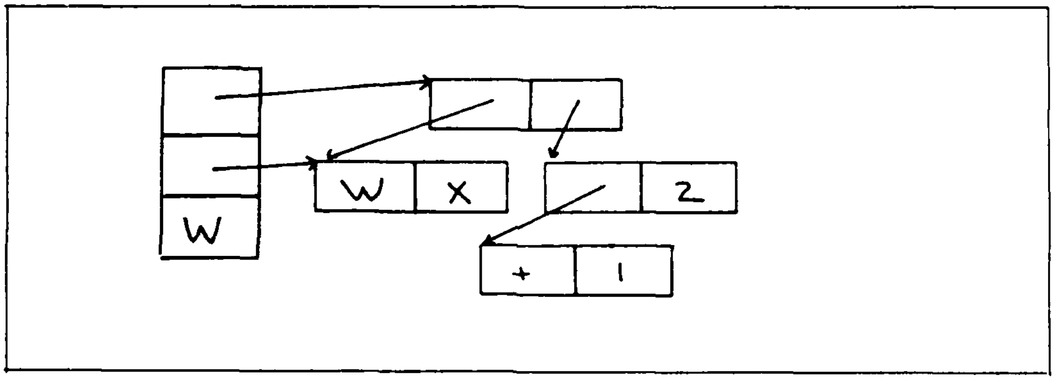
[24] The routine LNF-of-wff may not be used since it resets the stack of LASs before beginning Recall that LNF-of-subwff does not disturb the stack

(λ (?n) (× ?n ?n)) (+ 1 2)

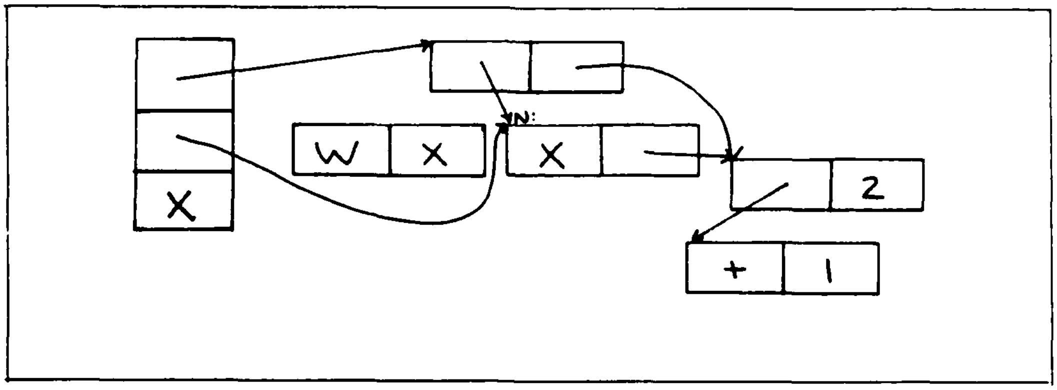
compiles.



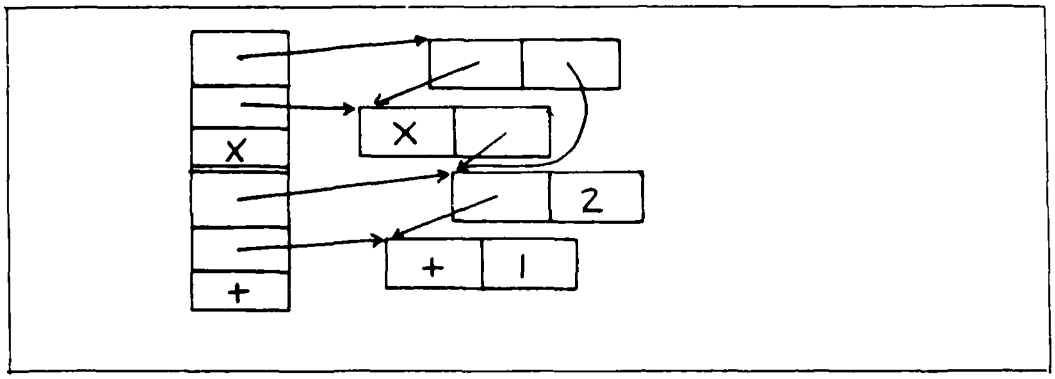
Before Calling LNF-of-wff
Figure 3.24



After Canonicalization
Figure 3.25

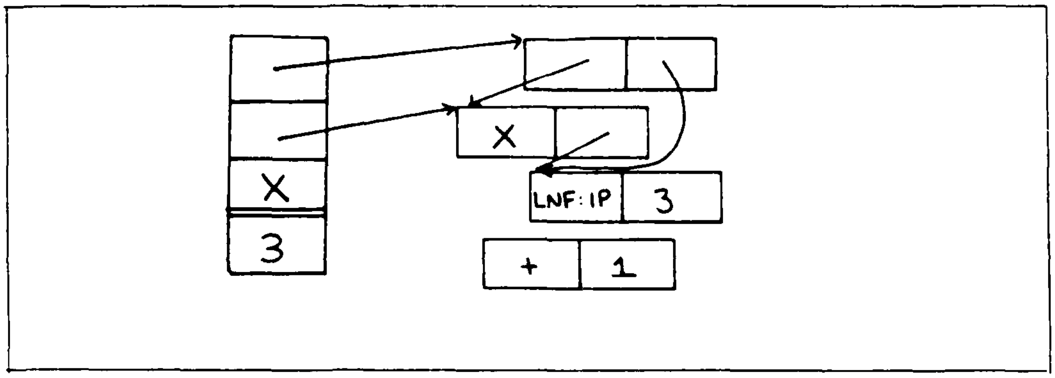


After W Reduction
Figure 3.26

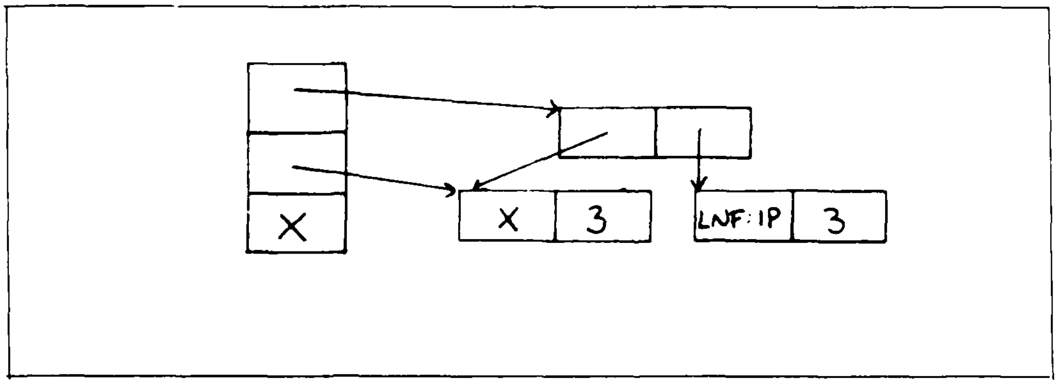


In the Middle of ×-reduce, Just Before + Reduction
Figure 3.27

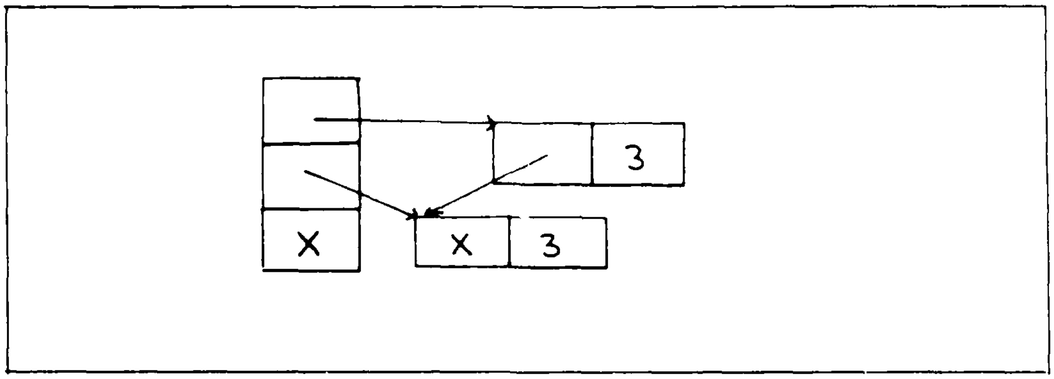
The routine +-reduce is identical to ×-reduce except for the expressions (× x y) and *RTP-×* which are replaced with (+ x y) and *RTP-+*, respectively.



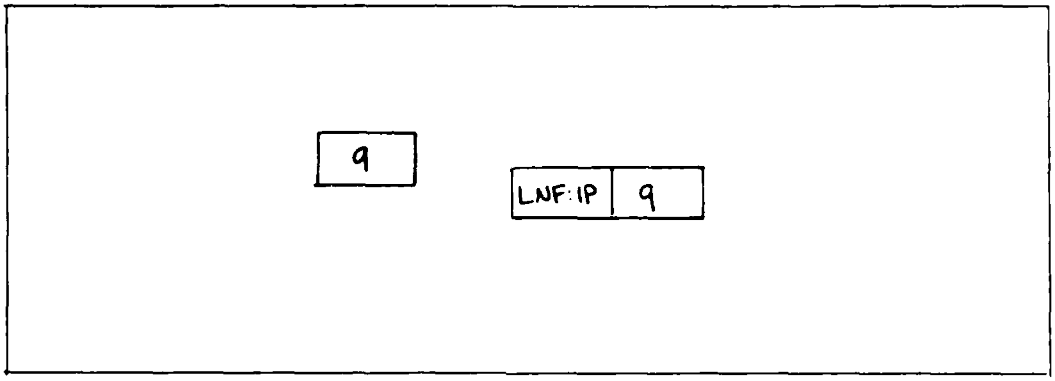
In the Middle of ×-reduce, Just After + Reduction.
LNF-of-subwff has not yet popped off the sum.
Figure 3.28



In the Middle of ×-reduce, Just After LNF-of-subwff Returns
Figure 3.29



Still Inside ×-reduce. Just After second Call on LNF-of-subwff Returns
Figure 3.30



After ×-reduce Returns
Figure 3.31

The significant aspects of LNF's runtime system have been presented. There are, of course, many more reduction routines; but their similarity to the routines just detailed obviates the need to present them here. It has been shown, mainly in pictures, that running an LNF program is nothing more than reducing an LNF-wff to lazy-normal form via the reduction rules of the LNF-calculus.

3.5. Displaying the Results

The function Display accepts LNF-wffs in lazy-normal form and displays their linearization on the screen. The user may elect to see the results of a computation (the reduced LNF-wff) in one of three formats:

- *Lazy-normal Form* — arguments of constructions and functions remain unreduced (the default)
- *Normal Form* — no redexes remain in the result
- *Normal Form of Members* — instead of a list's members being displayed surrounded by square brackets and separated by commas, just the (normal form of) each member is displayed

The user selects the display mode of choice by entering a directive (via the mouse). The system responds by changing its prompt (for the next LNf expression to be compiled, reduced, and displayed) to either:

- **LNF of** — (for lazy-normal form),
- **NF of** — (for normal form), or
- **NF of Members of** — (for normal form of members).

An example illustrates the effect the display mode has on the result. Suppose the LNF program to be run is:

TL [1,(+ 1 2),.,1,(× 2 2)].

In lazy-normal form mode the result displayed is:

[(+ 1 2),.,1,(× 2 2)],

in normal form mode the result displayed is:

[3,.,1,4],

and if the display mode is normal form of members, the following result is displayed:

3.14.

Display prints the normal form of an LNF-wff by, upon receiving an LNF-wff: $\mathbf{a}\,\mathbf{X}_1 \cdots \mathbf{X}_n$ in lazy-normal form, first printing **a**, then (recursively) calling (Display (LNF-of-wff $\mathbf{X}_i$)) for each i, $1 \le i \le n$. Thus, even for LNF-wffs which have no normal form, some output may be generated.

Observe that the display routine ensugars lists before displaying them — i.e. [1,2,3] is displayed rather than PAIR 1 (PAIR 2 (PAIR 3 [ ])). The display routine also knows about one other type of construction: the line. A line is construction of the form:

LINE (VEC $\mathbf{x}_0\ \mathbf{y}_0$) (VEC $\mathbf{x}_1\ \mathbf{y}_1$).

Lines are displayed by drawing the line from point $<\mathbf{x}_0,\mathbf{y}_0>$ to $<\mathbf{x}_1,\mathbf{y}_1>$ on the screen. If in normal form of members mode, a picture may be represented by a list of lines. A functional geometry program has been implemented in LNF and is displayed in Appendix C. The program is capable of creating an M.C. Escher print (following [Henderson 1982]) and producing "fractalized" pictures from existing pictures. The beauty of these programs is that the drawings are not side effects but normal-forms of their (very high level) description!

The routine Display is also capable of printing cyclic LNF-wffs of any kind. When displaying a non-list and Display encounters a cycle, it gives the LNF-wff (whose root it has seen before) a name and prints the name instead of the LNF-wff. When displaying a list, however, a name is not ascribed to the LNF-wff until the LNF-wff is seen for the third time, thus giving the user a better feeling for structure.

For example, the LNF-expression:

?x whererec [?x•?y] = [[1•?y]•[2•?x]]

which has the lazy-normal form:

PAIR 1 (ARG 2 (APP-TO-ARGS 2 (B (C′ PAIR (PAIR 1)) (PAIR 2)) ...)

is displayed (when in lazy-normal form mode) as:

[1•(ARG 2 (APP-TO-ARGS 2 (B (C′ PAIR (PAIR 1)) (PAIR 2)) E2023)]

but when in normal-form mode, is displayed as:

[1,2,1,2•P4825].

The names E2023 (E for Expression) and P4825 (P for Pair) are the system given names to the cyclic structures.

Functions and unknowns as well as constructions are displayed. A displayed function is just its linearized compiled code. For example, the squaring function:

λ (?n) (× ?n ?n)

is displayed as:

W ×.

Unknowns are displayed, simply, as linearized LNF-wffs.

3.6. Summary

LNF's experimental implementation has been described in fairly fine detail in this chapter. Special emphasis was placed on the compiler and the runtime system. The user interface to the system was only hinted at. Appendix D contains a recorded LNF session to give interested readers a feel for what *it's like to interact with* LNF.

Chapter 4 contains brief reviews of other work in this area, some comments on the relationship between this work and the author's, and some of the author's plans for the future of LNF.

Chapter 4

Summary, Related Work, and Future Plans

The author's work — having been detailed in chapters 1, 2, and 3 — is now summarized. In the section which summarizes LNF's implementation, brief discussions of other researchers' alternate approaches to compilation and runtime system organization are interspersed. Some of the author's plans for the future of the LNF language and its implementation have also been integrated into this synopsis.

4.1. Formal Aspects

Chapters 1 and 2 discuss the formal underpinnings of the LNF language. The content of these chapters is summarized in this section.

Following the presentation of two of the more famous reduction calculi: the λ-calculus ([Church 1941]) and the SKI-calculus ([Schönfinkel 1924]), the new concept of lazy-normal form is defined. The concept of lazy-normal form in the SKI-calculus is related to C.P. Wadsworth's concept of head-normal form ([Wadsworth 1971]) in the λ-calculus It is demonstrated (see Theorem 1.8) that an SKI-wff in lazy-normal form is an "outline" of the wff's normal form (if it exists) — i.e. its normal form will have the same initial atom and the same number of arguments. Theorem 1.8 also implies that an SKI-wff's normal form may be arrived at by first finding the wff's lazy-normal form and then applying this procedure recursively to its arguments. The implementation makes heavy use of both of these findings.

The ideas behind M. Schönfinkel's SKI-calculus, C.P. Wadsworth's graph oriented λ-G-calculus ([Wadsworth 1971]), and D.A. Turner's SASL implementation ([Turner 1979c]) are combined with the concept of lazy-normal form to produce a new deterministic combinator based graph and machine oriented reduction calculus, the SKI-G-calculus. This calculus is equivalent in power to the λ-calculus et al., but is much more directly and efficiently implementable. This is due primarily to the structure sharing properties of the SKI-G-wffs. Both garbage nodes and forwarding arcs (indirection pointers), concepts that are usually relegated to a calculus' implementation, are given formal definitions in this calculus.

The SKI-G-calculus still, however, is an inefficient model for a functional programming language's runtime system for the following two reasons. Translating (closed) λ-wffs into SKI-G-wffs (via a modified Schönfinkel abstraction algorithm) creates graphs of unacceptable size. Also, since the SKI-G-calculus is pure (i.e. free of numeric constants, numeric operators, conditional expressions, etc.), these familiar programming constructs must be represented in the calculus. The first problem is solved by using a different abstraction algorithm — one which produces much smaller SKI-G-wffs. This algorithm is based on the work presented in [Curry 1958], [Turner 1979a], and [Turner 1984a]. To solve the representation problem, new functors are defined (via new reduction rules) and a new type of atom is introduced: the constructor. The resulting calculus is called the LNF-calculus. It is this calculus upon which LNF's runtime system is based.

4.2. LNF's Implementation

The LNF language and its experimental implementation are detailed in Chapter 3. This section summarizes that implementation, discusses alternate methods for compiling and running functional programs, and presents some future plans for the implementation.

4.2.1. Compilation

The LNF language is a superset of the language of linearized LNF-wffs. In addition to the constructions, functions, and unknowns (linearized LNF-wffs, also called simple expressions) which are built from the atomic expressions via combination, the LNF language includes: lambda expressions, expressions having auxiliary declarations, list expressions, and conditional expressions. Lambda expressions may have bound expressions as formal parameters. Functions may be defined via order independent equations anywhere declarations are permitted. List expressions include both of the high level expression types which were introduced in D.A. Turner's KRC language ([Turner 1982a] and [Turner 1982b]): arithmetic sequences and ZF Expressions. Conditional expressions include case expressions having order independent cases. All LNF expressions have simple LNF expression (linearized LNF-wff) equivalents. The LNF compiler automates the transformation of LNF expressions into simple expressions for the user.

The compiler's main job is the elimination of bound expressions in favor of variable-free expressions. It accomplishes this via a generalized abstraction algorithm which, at its core, contains the Schönfinkel-Curry-Turner-Scheevel abstraction algorithm ([Turner 1984a]). Other FP language implementation projects which base their compiler on this abstraction algorithm include: D.A. Turner's SASL and Miranda languages ([Turner 1979c], [Turner 1984a], and [Turner 1984b]), Cambridge University's SKIM processor and its successor SKIM II ([Clarke 1980] and [Stoye 1984]), Burroughs Corporation's ARC-SASL language ([Richards 1984]), and Yale University's ALFL language ([Hudak 1984a], [Hudak 1984b], and [Hudak 1984c]).

Two similar FP language compilation algorithms, both different from the Schönfinkel et al. algorithm, are presented next. The first was developed by the Programming Methodology Group at Chalmers University for the language Lazy ML ([Augustsson 1984a], [Augustsson 1984b], [Johnsson 1984], [Kieburtz 1984], [Johnsson 1983], and [Johnsson 1981b]) and is called "lambda lifting". The other compilation algorithm was devised at Oxford University by R.J.M. Hughes ([Hughes 1982a] and [Hughes 1982b]) and is called "compilation via super-combinators". Both algorithms translate closed expressions involving abstractions, LET, and LETREC expressions into a set of reduction rules (each of which is independently compilable to a fixed program and defines a combinator to be used to reduce this one program) and an expression built up exclusively from atoms (constants and these tailored combinators) via combination.

The basic idea behind the lambda lifting and super-combinator approaches is to lift out to the outermost level all abstractions inside an expression. However, only closed abstractions may be "moved outside" without modification. For example, it is clear that the expression:

add1 ((λ x (* x x)) 30),

containing an interior closed abstraction, is equivalent to the expression:

(λ f (add1 (f 30))) (λ x (* x x))

containing no interior abstractions. The second expression may be viewed as the (singleton) set of reduction rules: {f x = * x x} and the abstraction-free combination: (add1 (f 30)). Before abstractions containing free variables may be "moved outside" they must be "closed up". This process of closing up such abstractions is where the two methods (lambda lifting and super-combinators) part ways. The lambda lifting approach closes up an abstraction containing free occurrence(s) of a variable **v** by passing **v** to it as argument and also adding **v** as a formal parameter. For example, the abstraction:

λ y (+ x x),

containing free occurrences of the variable x becomes the combination (containing only a closed abstraction):

(λ x (λ y (+ x x))) x.

The super-combinator approach specifies that the abstraction:

λ y (+ x x)

be transformed to this combination:

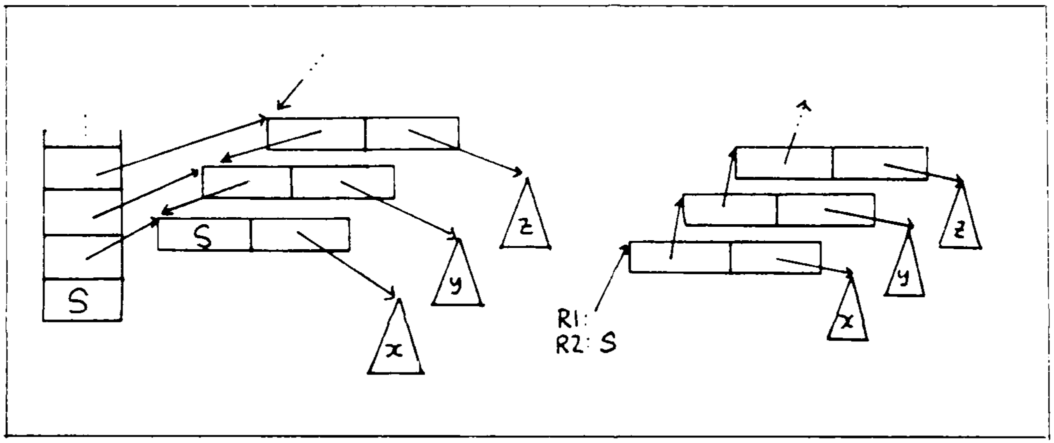
(λ s (λ y s)) (+ x x).

The difference, in general, is the following. Lambda lifting always abstracts away variables (the minimal free expressions) from the abstraction. The super-combinator approach abstracts away the maximal free expressions from the abstraction. Recall from Chapter 1 (in the discussion of Wadsworth's λ-G-calculus) that, sometimes, before some β-contractions could be performed, some parts of the operator (the abstraction) had to be copied. The parts that did not have to be copied were the abstraction's maximal free expressions. Arvind, in [Arvind 1984], points out that, in essence, Hughes' super-combinator abstraction algorithm is doing at compile time what Wadsworth's interpreter is doing at run time. The super-combinator compilation algorithm, by moving constant expressions outside of the bodies of abstractions, achieves full laziness. The lambda lifting approach is merely lazy.

After lambda lifting (or compilation to super-combinators), code must be generated from the set of reduction rules and the abstraction-free combination. Each reduction rule is compiled separately into a fixed program closely resembling the (hand-coded) functor specific reduction routines in the LNF runtime system. The abstraction-free combination is then reduced, in a runtime system organized along similar lines as LNF's, with the compiled reduction rules playing the part of the LNF's functor specific reduction routines.

4.2.2. The Runtime System

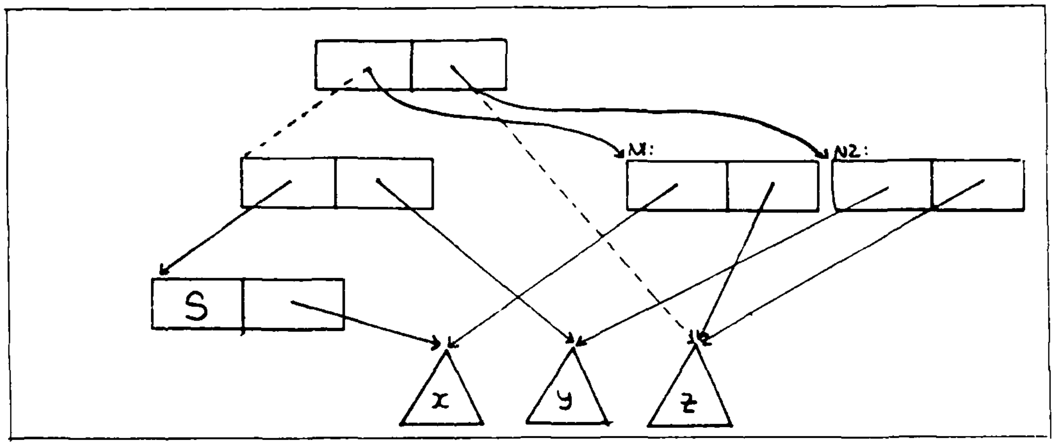
LNF's runtime system makes use of left ancestor stacks and hand-coded functor specific reduction routines. D.A. Turner's SASL and L. Augustsson's and T. Johnsson's Lazy ML projects both employ similar organizations. The SKIM, SKIM II, Miranda, and ARC-SASL projects use a scheme called "pointer reversal" in place of left ancestor stacks — in which the pointers along the left spine of the wff are reversed as they are encountered. Using the "pointer reversal" technique, the space taken up by the left ancestor stack is saved as this method requires only two registers — one to point to the wff's initial atom and one to point at the chain of reversed pointers. See the example below for a comparison of the two representations.



Left Ancestor Stack and Pointer Reversal Representations
Figure 4.1

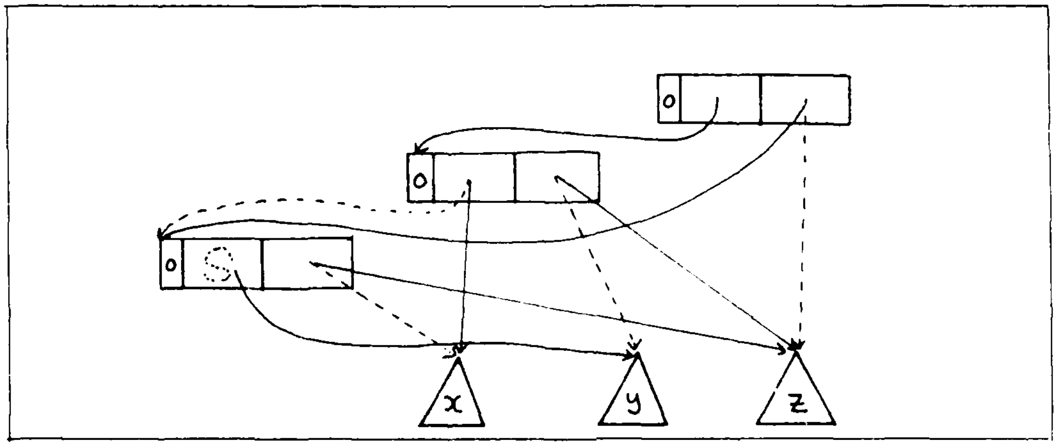
D.A. Turner credits, in [Turner 1984a], himself, A. Norman (SKIM and SKIM II), and M. Scheevel (ARC-SASL) with independently discovering this method. The author plans an experimental LNF implementation which uses pointer reversal in order to compare its performance with the left ancestor stack representation method.

The SKIM II runtime system performs some time and space saving optimizations, one of which has already been incorporated into the LNF system. After comparing two structures for equality (reducing a wff of the form: = **X Y**) and finding them equal, SKIM II's runtime system forwards one expression to the other. The two benefits arising from this operation are: (1) the cost of comparing the two wffs in the future will be minimal, and (2) many portions of the forwarded wff may become inaccessible and therefore eligible for reclamation. LNF's runtime system has borrowed this idea and put it to use. SKIM II's compiler, as mentioned above, is based on the Schönfinkel-Curry-Turner-Scheevel abstraction algorithm. Thus, the code it produces is similar to that produced by the LNF compiler — i.e LNF-wffs. The SKIM II *implementors* have added an extra field to the data structures which represent their graphs — a one bit reference count. The bit is turned on if more than one pointer points at the node — i.e. the node is shared. They employ this bit when reducing, for example, an S redex. In LNF, recall, an S redex is reduced as follows.



An LNF S Reduction
Figure 4.2

Observe that it requires two new combination cells (labeled n_1 and n_2) be allocated. The purpose of the one bit reference count is to avoid, whenever possible, allocating new cells. For example, if the node labeled 2 is not being shared before the reduction, then after the reduction this cell would be inaccessible — i.e. garbage. Instead of returning it to the heap at garbage collection time, the idea is to use it as one of the two required cells of the S reduction. If the node labeled 3 is also not being shared, then it could be used as the other "new" cell. An example of SKIM II S reduction follows.



A SKIM II S Reduction
Figure 4.3

In the above example all of the reference count bits are off. W.R. Stoye claims, in [Stoye 1984], "The results of applying this technique are spectacular — on average, about seventy percent of wasted cells are immediately reclaimed". It is planned that a future version of LNF will make use of this space saving scheme.

Other plans for the future of the LNF implementation include experimentation with:

- type inference ([Milner 1978], [Hindley 1969], [Damas 1982], and [Coppo 1980]), so as (1) to detect errors at compile time instead of waiting until runtime, and (2) to avoid the need for runtime type checking now present in many functor specific reduction routines.

- relaxing the rather artificial restrictions on the reduction rules defining functors like + and × which make them deterministic — i.e. allow them to reduce their arguments in parallel.

Appendix A

LNF-calculus' Linearized Reduction Rules

Each LNF-calculus reduction rule is displayed in this appendix. This presentation has them partitioned into two groups:

- Substantive Reduction Rules and
- Contextual Reduction Rules.

A.1. Substantive Reduction Rules

S	$S\,\mathbf{X\,Y\,Z} \rightarrow \mathbf{X\,Z}\,(\mathbf{Y\,Z})$
K	$K\,\mathbf{X\,Y} \rightarrow \mathbf{X}$
I	$I\,\mathbf{X} \rightarrow \mathbf{X}$
W	$W\,\mathbf{X\,Y} \rightarrow \mathbf{X\,Y\,Y}$
B	$B\,\mathbf{X\,Y\,Z} \rightarrow \mathbf{X}\,(\mathbf{Y\,Z})$
C	$C\,\mathbf{X\,Y\,Z} \rightarrow \mathbf{X\,Z\,Y}$
S′	$S'\,\mathbf{W\,X\,Y\,Z} \rightarrow \mathbf{W}\,(\mathbf{X\,Z})\,(\mathbf{Y\,Z})$
B′	$B'\,\mathbf{W\,X\,Y\,Z} \rightarrow \mathbf{W}\,(\mathbf{X}\,(\mathbf{Y\,Z}))$
C′	$C'\,\mathbf{W\,X\,Y\,Z} \rightarrow \mathbf{W}\,(\mathbf{X\,Z})\,\mathbf{Y}$
NUMBERP	NUMBERP **n** → TRUE NUMBERP **CFN** → FALSE, if $\underline{\text{CFN not a number}}$
+	$+\ \mathbf{n}\ \mathbf{m} \rightarrow \underline{\mathbf{n}+\mathbf{m}}$

×	× n m → n×m
-	- n m → n-m
DIV	DIV n m → n/m , if m≠0
IDIV	IDIV i j → integral quotient after i/j , if j≠0
REM	REM n m → remainder after n/m , if m≠0
EXP	EXP i j → the integer i^j , if j≥0 EXP i j → the float i^j , if j<0 EXP s i → the float s^i EXP n s → the float n^s
<	< n m → TRUE, if n<m < n m → FALSE, if n≥m
>	> n m → TRUE, if n>m > n m → FALSE. if n≤m
ADD1	ADD1 n → n+1
SUB1	SUB1 n → n-1
ZEROP	ZEROP n → n=0
BOOLEANP	BOOLEANP b → TRUE BOOLEANP CFN → FALSE, if CFN not a boolean
NOT	NOT TRUE → FALSE NOT FALSE → TRUE
OR	OR TRUE Y → TRUE OR FALSE b → b
AND	AND FALSE Y → FALSE AND TRUE b → b
HD	HD (PAIR X Y) → X
TL	TL (PAIR X Y) → Y
NULLP	NULLP [ ] → TRUE NULLP CFN → FALSE. if CFN≠[ ]
PAIRP	PAIRP (PAIR X Y) → TRUE PAIRP CFN → FALSE. if CFN not a pair

NTH	NTH 1 (PAIR **X Y**) → **X** NTH i (PAIR **X Y**) → NTH i-1 **Y**, if i≥1
APPEND	APPEND [ ] [ ] → [ ] APPEND [ ] **P** → **P** APPEND (PAIR **X Y**) **Z** → PAIR **X** (APPEND **Y Z**)
INTERLEAVE	INTERLEAVE [ ] **P** → **P** INTERLEAVE **P** [ ] → **P** INTERLEAVE (PAIR **X Y**) **P** → PAIR **X** (INTERLEAVE **P Y**)
FLATMAP	FLATMAP **X** [ ] → [ ] FLATMAP **X** (PAIR **Y Z**) → INTERLEAVE (**X Y**) (FLATMAP **X Z**)
ENUMERATE	ENUMERATE **X** → TURN [ ] **X**
TURN	TURN **X** [ ] → UP **X** [ ] [ ] TURN **X** (PAIR **Y Z**) → UP (PAIR **Y X**) [ ] **Z**
UP	UP [ ] **X Y** → DOWN **X** [ ] **Y** UP (PAIR [ ] **X**) **Y Z** → UP **X Y Z** UP (PAIR (PAIR $\mathbf{X}_1$ $\mathbf{X}_2$) **Y**) **W Z** → PAIR $\mathbf{X}_1$ (UP **Y** (PAIR $\mathbf{X}_2$ **W**) **Z**)
DOWN	DOWN [ ] [ ] [ ] → [ ] DOWN [ ] **P** [ ] → UP **P** [ ] [ ] DOWN (PAIR (PAIR $\mathbf{X}_1$ $\mathbf{X}_2$) **Y**) **Z W** → PAIR $\mathbf{X}_1$ (DOWN **Y** (PAIR $\mathbf{X}_2$ **Z**) **W**) DOWN [ ] **X** (PAIR [ ] **Y**) → TURN **X Y** DOWN [ ] **X** (PAIR (PAIR $\mathbf{Y}_1$ $\mathbf{Y}_2$) **Z**) → PAIR $\mathbf{Y}_1$ (TURN (PAIR $\mathbf{Y}_2$ **X**) **Z**)
MAP	MAP **X** [ ] → [ ] MAP **X** (PAIR **Y Z**) → PAIR (**X Y**) (MAP **X Z**)
MEMBER	MEMBER [ ] **X** → FALSE MEMBER (PAIR **X Y**) **Z** → IF (= **X Z**) TRUE (MEMBER **Y Z**)
COLLECT	COLLECT [ ] **X Y** → **Y** COLLECT (PAIR **X Y**) **W Z** → **W X** (COLLECT **Y W Z**)
FILTER	FILTER **X** [ ] → [ ] FILTER **X** (PAIR **Y Z**) → IF (**X Y**) (PAIR **Y** (FILTER **X Z**)) (FILTER **X Z**)
REM-DUPS	REM-DUPS **X** → REM-DUPS′ **X** [ ]

REM-DUPS′	REM-DUPS′ [ ] X → X REM-DUPS′ (PAIR X Y) Z → IF (MEMBER Z X) (REM-DUPS′ Y Z) (PAIR X (REM-DUPS′ Y Z))
FB	FB n m → PAIR n (FB′ $\underline{n+m}$ m), if $\underline{m \neq 0}$ FB n m → PAIR n (PAIR n ...), if $\underline{m=0}$
FB′	FB′ n m → PAIR n (FB′ $\underline{n+m}$ m)
FBT	FBT n m o → PAIR n (FBT′ $\underline{n+m}$ m o), if $\underline{(m>0 \text{ and } n \leq o) \text{ or } (m<0 \text{ and } n \geq o)}$ FBT n m o → [ ], if $\underline{(m>0 \text{ and } n>o) \text{ or } (m<0 \text{ and } n<o)}$ FBT n m o → PAIR n (PAIR n ...), if $\underline{m=0}$

FBT′	FBT′ n m o → PAIR n (FBT′ $\underline{n+m}$ m o),
	if $\underline{(m>0 \text{ and } n \leq o) \text{ or } (m<0 \text{ and } n \geq o)}$
	FBT′ n m o → [ ],
	if $\underline{(m>0 \text{ and } n>o) \text{ or } (m<0 \text{ and } [n<o)}$

Y	Y X → X (X (X ...))
=	= cf_1 cf_2 → $\underline{cf_1=cf_2}$ = CFN_1 CFN_2 → AND (= (OPERATOR CFN_1) (OPERATOR CFN_2)) (= (OPERAND CFN_1) (OPERAND CFN_2))
L	L cf CFN → TRUE, if $\underline{\text{NUM-ARGS[CFN]}>0}$ L CFN cf → FALSE, if $\underline{\text{NUM-ARGS[CFN]}>0}$ L cf_1 cf_2 → $\underline{cf_1 \text{ lexicographically less than } cf_2}$ L CFN_1 CFN_2 → OR (L (OPERATOR CFN_1) (OPERATOR CFN_2)) (AND (= (OPERATOR CFN_1) (OPERATOR CFN_2)) (L (OPERAND CFN_1) (OPERAND CFN_2))), if $\underline{CFN_1 \text{ and } CFN_2 \text{ are both combinations}}$

IF	IF TRUE X Y → X
	IF FALSE X Y → Y

UNKNOWNP	UNKNOWNP CFN → FALSE UNKNOWNP IRU → TRUE
FUNCTIONP	FUNCTIONP FN → TRUE FUNCTIONP CN → FALSE
FUNCTOR	FUNCTOR FN → $\underline{\text{INITIAL-ATOM[FN]}}$

CONSTRUCTIONP	CONSTRUCTIONP **CN** → TRUE CONSTRUCTIONP **FN** → FALSE
CONSTRUCTOR	CONSTRUCTOR $(\mathbf{c}\ \mathbf{X}_1 \quad \mathbf{X}_n) \rightarrow \mathbf{c}$
ARITY	ARITY **FN** → $\underline{\text{ARITY[INITIAL-ATOM[FN]] - NUM-ARGS[FN]}}$
NUM-ARGS	NUM-ARGS **CFN** → $\underline{\text{NUM-ARGS[CFN]}}$
ARG	ARG **i CFN** → $\underline{\text{ARG[i,CFN]}}$, if $\underline{1 \leq i \leq \text{NUM-ARGS[CFN]}}$
ATOMP	ATOMP **CFN** → $\underline{\text{NUM-ARGS[CFN]}=0}$
COMBINATIONP	COMBINATIONP **CFN** → $\underline{\text{NUM-ARGS[CFN]} > 0}$
OPERATOR	OPERATOR **CFN** → $\underline{\text{OPERATOR[CFN]}}$
OPERAND	OPERAND **CFN** → $\underline{\text{OPERAND[CFN]}}$
A-S-E	A-S-E $\mathbf{c}\ \mathbf{i}\ \mathbf{X}\ \mathbf{Y}\ (\mathbf{c}\ \mathbf{Z}_1 \quad \mathbf{Z}_i) \rightarrow \mathbf{X}$ A-S-E $\mathbf{c}_1\ \mathbf{i}\ \mathbf{X}\ \mathbf{Y}\ (\mathbf{c}_2\ \mathbf{Z}_1 \cdots \mathbf{Z}_j) \rightarrow \mathbf{Y}$, if $\underline{\mathbf{c}_1 \neq \mathbf{c}_2 \text{ or } \mathbf{i} \neq \mathbf{j}}$ A-S-E **c i X Y FN** → **Y**
A-S-E′	A-S-E′ $\mathbf{c}\ \mathbf{i}\ \mathbf{X}\ \mathbf{Y}\ (\mathbf{c}\ \mathbf{Z}_1 \cdots \mathbf{Z}_i) \rightarrow \mathbf{X}$ A-S-E′ $\mathbf{c}_1\ \mathbf{i}\ \mathbf{X}\ \mathbf{Y}\ (\mathbf{c}_2\ \mathbf{Z}_1 \cdots \mathbf{Z}_j) \rightarrow \mathbf{Y}$, if $\underline{\mathbf{c}_1 \neq \mathbf{c}_2 \text{ or } \mathbf{i} \neq \mathbf{j}}$ A-S-E′ **c i X Y FN** → **Y**
A-S	A-S $\mathbf{c}\ \mathbf{i}\ \mathbf{X}\ (\mathbf{c}\ \mathbf{Z}_1 \cdots \mathbf{Z}_i) \rightarrow \mathbf{X}\ \mathbf{Z}_1 \cdots \mathbf{Z}_i$
A-S′	A-S′ $\mathbf{c}\ \mathbf{i}\ \mathbf{X}\ (\mathbf{c}\ \mathbf{Z}_1 \quad \mathbf{Z}_i) \rightarrow \mathbf{X}\ \mathbf{Z}_1 \cdots \mathbf{Z}_i$
APP-TO-ARGS	APP-TO-ARGS **i X Y** → **X** (ARG 1 **Y**) ... (ARG **i Y**)

A.2. Contextual Reduction Rules

NUMBERP	NUMBERP **RDU** → NUMBERP **IMR**
+	+ **RDU Y** → + **IMR Y** + **n RDU** → + **n IMR**
×	× **RDU Y** → × **IMR Y** × **n RDU** → × **n IMR**

-	- **RDU Y** → - **IMR Y** - **n RDU** → - **n IMR**
DIV	DIV **RDU Y** → DIV **IMR Y** DIV **n RDU** → DIV **n IMR**
IDIV	IDIV **RDU Y** → IDIV **IMR Y** IDIV **i RDU** → IDIV **i IMR**
REM	REM **RDU Y** → REM **IMR Y** REM **n RDU** → REM **n IMR**
EXP	EXP **RDU Y** → EXP **IMR Y** EXP **n RDU** → EXP **n IMR**
<	< **RDU Y** → < **IMR Y** < **n RDU** → < **RDU IMR**
>	> **RDU Y** → > **IMR Y** > **n RDU** → > **n IMR**
ADD1	ADD1 **RDU** → ADD1 **IMR**
SUB1	SUB1 **RDU** → SUB1 **IMR**
ZEROP	ZEROP **RDU** → ZEROP **IMR**
BOOLEANP	BOOLEANP **RDU** → BOOLEANP **IMR**
NOT	NOT **RDU** → NOT **IMR**
OR	OR FALSE **RDU** → OR FALSE **IMR** OR **RDU Y** → OR **IMR Y**
AND	AND TRUE **RDU** → AND TRUE **IMR** AND **RDU Y** → AND **IMR Y**
HD	HD **RDU** → HD **IMR**
TL	TL **RDU** → TL **IMR**
NULLP	NULLP **RDU** → NULLP **IMR**
PAIRP	PAIRP **RDU** → PAIRP **IMR**
NTH	NTH **RDU Y** → NTH **IMR Y** NTH **i RDU** → NTH **i IMR**, if i>0
APPEND	APPEND **RDU Y** → APPEND **IMR Y**

INTERLEAVE	INTERLEAVE **RDU Y** → INTERLEAVE **IMR Y** INTERLEAVE **P RDU** → INTERLEAVE **P IMR**
FLATMAP	FLATMAP **X RDU** → FLATMAP **X IMR**
TURN	TURN **X RDU** → TURN **X IMR**
UP	UP (PAIR **RDU X**) **Y Z** → UP (PAIR **IMR X**) **Y Z** UP **RDU Y Z** → UP **IMR Y Z**
DOWN	DOWN [ ] **RDU** [ ] → DOWN [ ] **IMR** [ ] DOWN [ ] **Y RDU** → DOWN [ ] **Y IMR** DOWN [ ] **Y** (PAIR **RDU W**) → DOWN [ ] **Y** (PAIR **IMR W**) DOWN (PAIR **RDU X**) **Y Z** → DOWN (PAIR **IMR X**) **Y Z** DOWN **RDU Y Z** → DOWN **IMR Y Z**
MAP	MAP **X RDU** → MAP **X IMR**
MEMBER	MEMBER **RDU Y** → MEMBER **IMR Y**
COLLECT	COLLECT **RDU Y Z** → COLLECT **IMR Y Z**
FILTER	FILTER **X RDU** → FILTER **X IMR**
REM-DUPS′	REM-DUPS′ **RDU Y** → REM-DUPS′ **IMR Y**
FB	FB **RDU Y** → FB **IMR Y** FB **n RDU** → FB **n IMR**
FBT	FBT **RDU Y Z** → FBT **IMR Y Z** FBT **n RDU Z** → FBT **n IMR Z** FBT **n m RDU** → FBT **n m IMR**
=	= **RDU Y** → = **IMR Y** = **CFN RDU** → **CFN IMR**
L	L **RDU Y** → L **IMR Y** L **CFN RDU** → L **CFN IMR**
IF	IF **RDU X Y** → IF **IMR X Y**
UNKNOWNP	UNKNOWNP **RDU** → UNKNOWNP **IMR**
FUNCTIONP	FUNCTIONP **RDU** → FUNCTIONP **IMR**
FUNCTOR	FUNCTOR **RDU** → FUNCTOR **IMR**
CONSTRUCTIONP	CONSTRUCTIONP **RDU** → CONSTRUCTIONP **IMR**

CONSTRUCTOR	CONSTRUCTOR **RDU** → CONSTRUCTOR **IMR**
ARITY	ARITY **RDU** → ARITY **IMR**
NUM-ARGS	NUM-ARGS **RDU** → NUM-ARGS **IMR**
ARG	ARG **RDU Y** → ARG **IMR Y** ARG **i RDU** → ARG **i IMR**
ATOMP	ATOMP **RDU** → ATOMP **IMR**
COMBINATIONP	COMBINATIONP **RDU** → COMBINATIONP **IMR**
OPERATOR	OPERATOR **RDU** → OPERATOR **IMR**
OPERAND	OPERAND **RDU** → OPERAND **IMR**
A-S-E	A-S-E **c i X Y RDU** → A-S-E **c i X Y IMR**
A-S	A-S **c i X RDU** → A-S **c i X IMR**

Appendix B

BNF-like Description of LNF Expressions

Sprinkled throughout the formal description of the language are examples of well-formed LNF-exps. The description makes use of the following conventions:

- UPPERCASE names denote syntactic categories.
- The symbol ∪ denotes category union.
- Lowercase names are concrete syntax.
- <..> denotes an optional item.
- <..>* denotes 0 or more items.
- <..>+ denotes 1 or more items.

LNF-EXP ::= SIMPLE-EXP ∪ LAMBDA-EXP ∪
WITH-AUX-DECL-EXP ∪ LIST-EXP ∪ CONDITIONAL-EXP

SIMPLE-EXP ::= ATOM ∪ COMBINATION ∪ (LNF-EXP)
ATOM ::= CONSTRUCTOR ∪ FUNCTOR ∪ VARIABLE
CONSTRUCTOR ::= ZETALISP-SYMBOL
COMBINATION ::= LNF-EXP LNF-EXP

All VARIABLE occurrences must be bound occurrences.

EXAMPLES: (of SIMPLE expressions)

39882736

(((23)))

flat-Tire

pair 2 4

S f g

(if TRUE then 4 3)

+ 4934732984

× (minus 2432) (- box bag)

LAMBDA-EXP ::= λ (<BE>$^{+}$) LNF-EXP
BE ::= VARIABLE ∪ CONSTRUCTED-BE
VARIABLE ::= NAMED-VARIABLE ∪ ANONYMOUS-VARIABLE
NAMED-VARIABLE ::= ?ZETALISP-SYMBOL
ANONYMOUS-VARIABLE ::= ?
CONSTRUCTED-BE ::= CONSTRUCTOR <BE>* ∪ LIST-BE
LIST-BE ::= [ ] ∪ [BE<,BE>*<•BE>]

The list of formal parameters may contain only one occurrence of any one (non anonymous) variable.

EXAMPLES: (of LAMBDA expressions)

λ (?x) (+ ?x ?x)

λ ([?x•?y] ?p) (or (?p ?x) (or-list (map ?p ?y)))

λ ((ds ?f1 ?f2 ?f3)) (?f3 (+ ?f1 ?f2))

λ (0) 1

WITH-AUX-DECL-EXP ::= WHERE-EXP ∪ WHEREREC-EXP ∪ WHERE*-EXP
WHERE-EXP ::= LNF-EXP where DECLARATION <& DECLARATION>*
WHEREREC-EXP ::= LNF-EXP whererec DECLARATION <& DECLARATION>*
WHERE*-EXP ::= LNF-EXP where* DECLARATION <; DECLARATION>*
DECLARATION ::= SIMPLE-DECLARATION ∪ FUNCTION-DECLARATION
SIMPLE-DECLARATION ::= VARIABLE = LNF-EXP ∪ (CONSTRUCTED-BE) = LNF-EXP
FUNCTION-DECLARATION ::= FUNCTION-EQN ∪ EQUATION-SET
FUNCTION-EQN ::= ZETALISP-ATOM <BE>+ = LNF-EXP
EQUATION-SET ::= {FUNCTION-EQN | <|FUNCTION-EQN>+}

Each FUNCTION-EQN in the set must be headed by the same ZETALISP-ATOM.

EXAMPLES: (of WHERE, WHEREREC, and WHERE* expressions)

```
 (− ?x ?y) where ?x = 3 & ?y = 4

 ?p1 whererec ?p1 = [1•?p2] & ?p2 = [2•?p1]

 (× ?x ?y) where* ?x = 3 ; ?y = (factorial ?x)

 (thrice double 5) where
   thrice ?f ?x = ?f (?f (?f ?x)) &
   double ?x = × 2 ?x

 (+ ?x ?y) where (tree ?x ? ?y) = some-tree

 (factorial 10) whererec
   factorial ?n = (if (zerop ?n) then 1
                  else (× ?n (factorial (sub1 ?n))))

 (app [1,2,3] list) whererec
   {app [ ] ?z = ?z |
    app [?x•?r] ?z = [?x•(app ?r ?z)]}
```

LIST-EXP ::= EXPLICIT-LIST-EXP ∪ ARITH-SEQ-EXP ∪ IMPLICIT-LIST-EXP
EXPLICIT-LIST-EXP ::= [ ] ∪ [LNF-EXP<,LNF-EXP>*<•LNF-EXP>]
ARITH-SEQ-EXP ::= [LNF-EXP<,LNF-EXP>...<,LNF-EXP>]
IMPLICIT-LIST-EXP ::= FOR-EACH-EXP ∪ TURNER-LIST-EXP
TURNER-LIST-EXP ::= [LNF-EXP|GENERATOR<<;GUARD><;GENERATOR>>*]
FOR-EACH-EXP ::= for-each GENERATOR FOR-EACH-CLAUSE
FOR-EACH-CLAUSE ::= and-for-each GENERATOR FOR-EACH-CLAUSE ∪
 such-that GUARD FOR-EACH-CLAUSE ∪
 instantiate LNF-EXP
GUARD ::= LNF-EXP
GENERATOR ::= BE∈LNF-EXP

EXAMPLES: (of LIST expressions)

```
[1,2,3,4,5]

[flat,2,tire,1•23]

[a•b]

[a,b,c•d]

[1,..]

[10,10,..]

[1,3, .]

[0,-1, .]

[2,4,..,100]

[1, .,1000]

[(* 10 ?x)|?x∈[1,..]]

[[?x•?y]|?x∈[1,..,5];(odd ?x);?y∈[100,101]]

[(+ ?x ?y)|[?x•?y]∈(zip [1,..,10] [100,..,110])]

for-each ?x∈[1,..]
  instantiate (× 10 ?x)

for-each ?x∈[1,..,5]
  such-that (odd ?x)
  and-for-each ?y∈[100,101]
  instantiate [?x•?y]

for-each [?x•?y]∈(zip [1,...,10] [100,...110])
  instantiate (+ ?x ?y)
```

CONDITIONAL-EXP ::= IF-EXP ∪ CASE-EXP
IF-EXP ::= if LNF-EXP <then> LNF-EXP <else> LNF-EXP
CASE-EXP ::= case LNF-EXP in BE → LNF-EXP <| BE → LNF-EXP>* endcase

EXAMPLES: (of CONDITIONAL expressions)

```
if (odd num) 2 3

if (odd num) then 2 3

if (odd num) 2 else 3

if (odd num) then 2 else 3

case a-tree in
   (tree ?left ?root ?right) → (append (leaves ?left) [?root•(leaves ?right)]) |
   nulltree → [ ]
endcase

case (leaves big-tree) in
  [?•?rest] → (add1 (len ?rest)) |
  [ ] → 0
endcase
```

Appendix C

Examples of LNF Function Definitions

The format of the definitions is as follows. To define the symbol **S** to be the expression **E**, enter:

(define **S E**).

To define the function **F** with formal parameters $\mathbf{A}_1, \ldots, \mathbf{A}_n$ and body **B**, enter either:

(define (**F** $\mathbf{A}_1 \cdots \mathbf{A}_n$) **B**)

or

(define **F** (λ ($\mathbf{A}_1 \cdots \mathbf{A}_n$) **B**)).

To define the function **G** (via $\mathbf{B}_m$ equations), where the $\mathbf{B}_i$th equation has formal parameters $\mathbf{A}_{i_1}, \ldots, \mathbf{A}_{i_n}$ and body $\mathbf{B}_i$:

(define (**G** $\mathbf{A}_{1_1} \cdots \mathbf{A}_{1_n}$) $\mathbf{B}_1$

...

(**G** $\mathbf{A}_{m_1} \cdots \mathbf{A}_{m_n}$) $\mathbf{B}_m$).

A semicolon signals the beginning of a comment. A comment ends at the end of a line.

A sample LNF session, making use of many of these functions, has been recorded and placed in Appendix D.

C.1. Some Utility Functions

```
;;; Returns the first n elements of a nonempty list
(define (first ?n [?x•?r])
  (if (zerop ?n)
     then [ ]
   else [?x•(first (sub1 ?n) ?r)]))
```

```
;;; Returns absolute value of x
(define (abs ?x) (if (< 0 ?x) ?x (minus ?x)))

;;; Returns n+m modulo mod
(define (plus-mod ?mod ?n ?m)
  (rem (+ ?n ?m) ?mod))

;;; Places first element of nonempty list at the rear.
(define (rotate [?x•?r])
  (append ?r [?x]))

;;; Exchanges first and second elements of a list.
(define (exchange [?x1,?x2•?r])
  [?x2,?x1•?r])

;;; Reverses a nonempty list.
(define (reverse [?x•?r])
  (if (nullp ?r)
     then [?x]
   else (append (reverse ?r) [?x])))
```

C.2. Closing Up "Sets" Under Laws

```
;;; These next three definitions are LNF versions of functions
;;; written by D.A. Turner.  They appear in [Turner 1981a].

;;; Returns a set (represented as a list w/o duplicates), which
;;; is ?set closed up under the operations (LNF functions) in the
;;; list ?laws.
(define (closure-under-laws ?laws ?set)
  (append ?set (closure1 ?laws ?set ?set)))

;;; Returns the "set" which is ?set2 closed under ?laws
;;; minus the "set" ?set1.
(define (closure1 ?laws ?set1 ?set2)
  (closure2
    ?laws
    ?set1
    ;; mkset removes duplicate elements from a list
    (mkset [?a | ?law ∈ ?laws ;
                 ?a ∈ (map ?law ?set2) ;
                  (not (member ?set1 ?a))])))
```

```
;;; Returns the "set" which is ?set2 closed under ?laws
;;; minus the "set" ?set1.
(define (closure2 ?laws ?set1 ?set2)
  (if (nullp ?set2)
     then [ ]
    else (append
           ?set2
           (closure1 ?laws (append ?set1 ?set2) ?set2))))

;;; SOME INTERESTING SETS

;;; The Naturals modulo ?mod — defined as the set [0] closed
;;; under the "successor modulo ?mod" function
(define (naturals-modulo-n ?mod)
  (closure-under-laws [plus-mod ?mod 1] [0]))

;;; The Naturals — the set [0] closed under the successor
;;; function.
(define naturals
  (closure-under-laws [add1] [0]))

;;; The Integers — the set [0] closed under the successor and
;;; predecessor functions.
(define integers-rep1
  (closure-under-laws [add1,sub1] [0]))

;;; The Integers (again) — the set [0] closed under the
;;; predecessor and the absolute value functions.
(define integers-rep2
  (closure-under-laws [abs,sub1] [0]))

;;; The even Integers — the set [0] closed under the
;;; "decrement by 2" and the absolute value functions.
(define even-integers
  (closure-under-laws [abs,(λ (?x) (- ?x 2))] [0]))

;;; The powers of ?n — the set [1] closed under the
;;; "multiply by ?n" function.
(define (powers-of ?n)
  (closure-under-laws [* ?n] [1]))

;;; A STRANGE set — the set [[0]] (whose only element is a set)
;;; closed under the function which closes sets under the
;;; "successor modulo ?mod" function.
(define (higher-order-example-mod ?mod)
  (closure-under-laws
     [closure-under-laws [plus-mod ?mod 1]]
     [[0]]))
```

```
;;; The set of all permutations of ?list.
(define (perms ?list)
  (closure-under-laws [exchange,rotate,reverse] [?list]))
```

C.3. Geometric Sequences and Series

```
;;; Returns the geometric sequence [a,ax,ax^2,ax^3,...].
(define (g-seq ?a ?x)
  (g-seq-from-n ?a ?x 0))

;;; Returns the geometric sequence tail [ax^n ,ax^n+1,...].
(define (g-seq-from-n ?a ?x ?n)
  ([(× ?a (exp ?x ?n))•(g-seq-from-n ?a ?x (add1 ?n))]))

;;; Returns the infinite series corresponding to the given
;;; infinite sequence.
(define (series [?x•?rest])
  ([?x•series1 [?x•?rest]]))

;;; Helper function for series.
(define (series1 [?x1,?x2•?rest])
  ([?z•series1 [?z•?rest]] where ?z = (+ ?x1 ?x2)))

;;; Returns TRUE when applied to a convergent geometric series.
(define (convergent-g-series [?x1,?x2•?rest])
  ((and (< -1 ?x) (< ?x 1))
   where ?x = (div (- ?x2 ?x1) ?x1)))

;;; Returns the limit of a convergent geometric series.
(define (limit-g-series [?x1,?x2•?rest])
  ((div ?x1 (- ?x 1))
   where ?x = (div (- ?x2 ?x1) ?x1)))

;;; Returns a pair [n•x] where x is the nth element in
;;; the series and is the first element to be within epsilon of
;;; the series' limit.
(define (first-close-to-limit ?series ?epsilon)
  (first-close-to-limit1
    ?series
    ?epsilon
    (limit-g-series ?series)
    0))
```

```
;;; Same as above except that the limit has already been
;;; determined and the first n elements are not within epsilon
;;; of the limit.
(define
  (first-close-to-limit1 [?xn+1•?rest] ?epsilon ?limit ?n)
  ((if (within-epsilon ?xn+1 ?limit ?epsilon)
      then [?n-plus-one•?xn+1]
    else (first-close-to-limit1
           ?rest
           ?epsilon
           ?limit
           ?n-plus-one))
   where ?n-plus-one = (add1 ?n)))

;;; Returns TRUE iff x1 is within epslion of x2.
(define (within-epsilon ?x1 ?x2 ?epsilon)
  ((< (?abs ?diff) ?epsilon)
   where ?diff = (- ?x1 ?x2) &
        ?abs ?num = if (> ?num 0) ?num (minus ?num)))
```

C.4. Functional Geometry

An LNF implementation of Peter Henderson's "Functional Geometry" ([Henderson 1982]) follows. There is one big difference between Henderson's implementation and the author's. For Henderson, pictures are data structures, but in the LNF implementation, pictures are functions. A picture is a function, which when applied to three arguments, each of which is a vector of the form: VEC x y, becomes a plottable picture. A plottable picture is simply a list of plottable lines, each taking the form. LINE (VEC x0 y0) (VEC x1 y1). LINE and VEC are contructors. The suite of functions which implements these ideas follows.

```
;;; Vector addition.
(define (vec+vec (vec ?x0 ?y0) (vec ?x1 ?y1))
  (vec (+ ?x0 ?x1) (+ ?y0 ?y1)))

;;; Scalar-vector multiplication.
(define (scalar*vec ?n (vec ?x ?y))
  (vec (× ?n ?x) (× ?n ?y)))

;;; The Basic Functions:

;;; Implements PH's nil (the empty picture), i.e. a function
;;; of arity 3 which, when applied, ignores its arguments and
;;; returns the empty list.
(define (empty-pic ? ? ?) [ ])
```

```
;;; Implements PH's: plot(grid(m,n,s),a-vec,b-vec,c-vec)
;;; (grid m n segs) → picture
;;; (grid m n segs avec bvec cvec) → plottable-picture
;;; NOTE: plot is unnecessary in this implementation.
(define (grid ?m ?n ?segments ?a-vec ?b-vec ?c-vec)
 (for-each (segment ?x0 ?y0 ?x1 ?y1) in ?segments
   instantiate
   (line (vec+vec ?a-vec
             (vec+vec (scalar*vec (div ?x0 ?m) ?b-vec)
                   (scalar*vec (div ?y0 ?n) ?c-vec)))
       (vec+vec ?a-vec
             (vec+vec (scalar*vec (div ?x1 ?m) ?b-vec)
                   (scalar*vec (div ?y1 ?n) ?c-vec))))))

;;; Implements PH's: plot(flip(p),a-vec,b-vec,c-vec)
;;; (flip picture) → picture
;;; (flip picture avec bvec cvec) → plottable-picture
(define (flip ?pic ?a-vec ?b-vec ?c-vec)
 (?pic (vec+vec ?a-vec ?b-vec)
     (scalar*vec -1 ?b-vec)
     ?c-vec))

;;; Implements PH's: plot(rot(p),a-vec,b-vec,c-vec)
;;; (rot picture) → picture
;;; (rot picture avec bvec cvec) → plottable-picture
(define (rot ?pic ?a-vec ?b-vec ?c-vec)
 (?pic (vec+vec ?a-vec ?b-vec)
     ?c-vec
     (scalar*vec -1 ?b-vec)))

;;; Implements PH's: plot(overlay(p,q),a-vec,b-vec,c-vec)
;;; (overlay picture picture) → picture
;;; (overlay picture picture avec bvec cvec) → plottable-picture
(define (overlay ?pic1 ?pic2 ?a-vec ?b-vec ?c-vec)
  (append (?pic1 ?a-vec ?b-vec ?c-vec)
      (?pic2 ?a-vec ?b-vec ?c-vec)))

;;; Implements PH's: plot(beside(m,n,p,q),a-vec,b-vec,c-vec)
;;; (beside n m picture picture) → picture
;;; (beside n m picture picture avec bvec cvec) →
;;;    plottable-picture
(define (beside ?m ?n ?left-pic ?right-pic ?a-vec ?b-vec ?c-vec)
  ((append (?left-pic ?a-vec ?scaled-b-vec ?c-vec)
       (?right-pic (vec+vec ?a-vec ?scaled-b-vec)
               (scalar*vec (div ?n (+ ?m ?n)) ?b-vec)
               ?c-vec))
  where ?scaled-b-vec = (scalar*vec (div ?m (+ ?m ?n)) ?b-vec)))
```

```
;;; Implements PH's: plot(above(m,n,p,q),a-vec,b-vec,c-vec)
;;; (above n m picture picture) → picture
;;; (above n m picture picture avec bvec cvec) → plottable-picture
(define (above ?m ?n ?top-pic ?bot-pic ?a-vec ?b-vec ?c-vec)
 ((append (?top-pic
           (vec+vec ?a-vec ?scaled-c-vec)
           ?b-vec
           (scalar*vec (div ?m (+ ?m ?n)) ?c-vec))
          (?bot-pic ?a-vec ?b-vec ?scaled-c-vec))
   where ?scaled-c-vec = (scalar*vec (div ?n (+ ?m ?n)) ?c-vec)))

;;; PH's quartet
;;; (quartet picture picture picture picture) → picture
(define (quartet ?p1 ?p2 ?p3 ?p4)
 (above 1 1 (beside 1 1 ?p1 ?p2) (beside 1 1 ?p3 ?p4)))

;;; PH's cycle
;;; (cycle picture) → picture
(define (cycle ?pic)
 ((quartet ?pic
           (rot ?rot-rot-pic)
           ?rot-pic
           ?rot-rot-pic)
    where* ?rot-pic = (rot ?pic) ;
           ?rot-rot-pic = (rot ?rot-pic)))

;;; Some Example Pictures From PH's Paper:

;;; PH's man
(define man
 (grid 14 20
      [segment 6 10 0 10, segment 0 10 0 12,
       segment 0 12 6 12, segment 6 12 6 14,
       segment 6 14 4 16, segment 4 16 4 18,
       segment 4 18 6 20, segment 6 20 8 20,
       segment 8 20 10 18, segment 10 18 10 16,
       segment 10 16 8 14, segment 8 14 8 12,
       segment 8 12 10 12, segment 10 12 10 14,
       segment 10 14 12 14, segment 12 14 12 10,
       segment 12 10 8 10, segment 8 10 8 8,
       segment 8 8 10 0, segment 10 0 8 0,
       segment 8 0 7 4, segment 7 4 6 0,
       segment 6 0 4 0, segment 4 0 6 8,
       segment 6 8 6 10]))

;;; PH's FatBoy
(define fatboy (above 1 1 empty-pic man))
```

```
;;; PH's Boy
(define boy (beside 1 1 fatboy empty-pic))

;;; Components Making up Escher Print:

;;; The next 6 pictures are the basic buiding blocks of the print.

;;; PH's p, figure 18 in paper
(define mce-p
  (grid 36 36
        [;; left eye
         segment 0 7 6 9, segment 6 9 0 18, segment 0 18 0 7,
         ;; line between eyes
         segment 13 0 9 9,
         ;; right eye
         segment 9 12 9 23, segment 9 23 16 14, segment 16 14 9 12,
         ;; side of head
         segment 24 0 22 9, segment 22 9 18 18,
         segment 18 18 9 30, segment 9 30 0 36,
         ;; top of tail
         segment 0 36 13 34, segment 13 34 18 36,
         segment 18 36 26 27, segment 26 27 36 27,
         ;; line in tail
         segment 18 27 36 23,
         ;; bottom of tail
         segment 18 18 27 21, segment 27 21 36 18,
         ;; tiny line in upper right
         segment 32 36 36 34,
         ;; next one down
         segment 27 36 29 34, segment 29 34 36 32,
         ;; and the next
         segment 22 36 26 32, segment 26 32 36 29,
         ;; first line below tail
         segment 20 14 27 16, segment 27 16 36 14,
         ;; the next
         segment 22 9 29 11, segment 29 11 36 9,
         ;; and, finally, the last
         segment 24 0 31 5, segment 31 5 36 5]))
```

```
;;; PH's q, figure 19 in paper
(define mce-q
  (grid 36 36
        [;; left side of fish
         segment 0 27 7 29, segment 7 29 11 31.
         segment 11 31 16 34, segment 16 34 18 36.
         ;; line in middle of fish
         segment 0 23 16 25,
         ;; left edge
         segment 0 27 0 36, segment 0 0 0 18,
         ;; right side of fish
         segment 0 18 9 16, segment 9 16 13 16,
         segment 13 16 27 22, segment 27 22 36 36.
         ;; leftmost line above fish
         segment 4 36 7 29,
         ;; next one
         segment 9 36 11 31,
         ;; rightmost line above fish
         segment 14 36 16 34,
         ;; left eye
         segment 18 34 25 34, segment 25 34 20 30,
         segment 20 30 18 34,
         ;; right eye
         segment 20 27 27 27, segment 27 27 22 23,
         segment 22 23 20 27,
         ;; right side of tail
         segment 36 36 34 22, segment 34 22 36 18,
         segment 36 18 29 9, segment 29 9 27 0,
         ;; three lines to the right of the tail
         segment 29 0 36 14, segment 32 0 36 9,
         segment 34 0 36 4,
         ;; line in tail
         segment 32 25 23 0,
         ;; four lines left of tail (left to right)
         segment 5 0 9 11, segment 9 11 9 16,
         segment 9 0 13 11, segment 13 11 13 16.
         segment 14 0 18 13, segment 18 13 18 18.
         segment 18 0 22 14, segment 22 14 22 20]))
```

```
;;; PH's r, figure 20 in paper
(define mce-r
  (grid 36 36
        [;; top of fish
         segment 24 36 27 28, segment 27 28 36 18,
         ;; bottom of fish
         segment 0 36 4 27, segment 4 27 10 22,
         segment 10 22 17 18, segment 17 18 31 14,
         segment 31 14 36 9,
         ;; line thru fish
         segment 13 36 25 23, segment 25 23 36 14,
         ;; lines above fish
         segment 27 28 36 36, segment 29 30 36 23,
         segment 31 32 36 28,segment 33 34 36 32,
         ;; bottom semi-horizontal lines
         segment 2 2 8 0, segment 4 4 18 0, segment 7 7 18 4,
         segment 18 4 27 0, segment 10 11 27 7, segment 27 7 36 0,
         ;; lower diagonal lines
         segment 0 0 17 18, segment 0 8 10 22,
         segment 0 18 4 27, segment 0 27 2 32]))
```

```
;;; PH's s, figure 21 paper
(define mce-s
  (grid 36 36
        [;; left fish
         segment 18 36 16 30, segment 16 30 16 23,
         segment 16 23 16 18, segment 16 18 18 14,
         segment 18 14 23 9, segment 23 9 36 0,
         ;; line in fish
         segment 23 36 25 23,
         ;; right fish
         segment 27 36 30 30, segment 30 30 32 25,
         segment 32 25 34 21, segment 34 21 36 18,
         ;; right eye
         segment 29 16 34 18, segment 34 18 34 11,
         segment 34 11 29 16,
         ;; left eye
         segment 22 14 27 16, segment 27 16 27 9,
         segment 27 9 22 14,
         ;; lines right of fish
         segment 30 30 36 32, segment 32 25 36 27,
         segment 34 2T 36 22,
         ;; bottom hump
         segment 0 0 9 5, segment 9 5 17 5, segment 17 5 36 0,
         ;; next up
         segment 0 9 4 2, segment 0 14 16 9,
         segment 0 18 18 14, segment 0 23 16 18,
         segment 0 28 16 23, segment 0 32 16 30,
         ;; top border lines
         segment 0 36 18 36, segment 27 36 36 36]))

;;; PH's t, figure 22 in paper
(define mce-t
  (quartet mce-p mce-q mce-r mce-s))

;;; PH's u, figure 23 in paper
(define mce-u
  (cycle (rot mce-q)))

;;; The remaining functions are used to combine the basic building
;;; blocks into the Escher print.

(define side1
  (quartet empty-pic empty-pic (rot mce-t) mce-t))

(define side2
  (quartet side1 side1 (rot mce-t) mce-t))

(define corner1
  (quartet empty-pic empty-pic empty-pic mce-u))
```

```
(define corner2
  (quartet corner1 side1 (rot side1) mce-u))

(define pseudocorner
  (quartet corner2 side2 (rot side2) (rot mce-t)))

(define pseudolimit
  (cycle pseudocorner))

(define (nonet ?p1 ?p2 ?p3 ?p4 ?p5 ?p6 ?p7 ?p8 ?p9)
  (above 1 2
        (beside 1 2 ?p1 (beside 1 1 ?p2 ?p3))
        (above 1 1
              (beside 1 2 ?p4 (beside 1 1 ?p5 ?p6))
              (beside 1 2 ?p7 (beside 1 1 ?p8 ?p9)))))

(define corner
  ((nonet
     corner2    side2      side2
     ?rot-side2  mce-u      ?rot-mce-t
     ?rot-side2  ?rot-mce-t (rot mce-q))
   where ?rot-side2 = (rot side2) &
         ?rot-mce-t = (rot mce-t)))

(define squarelimit
  (cycle corner))

;;; Entering "squarelimit (vec 50 50) (vec 500 0) (vec 0 500)"
;;; at the LNF prompt "NF of Members " produces the Escher print.

;;; The functions below "fractalize" pictures.

;;; Given a natural number n, a fractal-function, and a picture,
;;; the next function applies the fractal-function n times to
;;; the picture (actually, it is applied to each of the picture's
;;; lines) — producing a fractalized picture.
(define (fractalize ?n ?fractal-fn ?pic ?a-vec ?b-vec ?c-vec)
  ((if (zerop ?n)
      then ?plottable-picture
    else (fractalize1
          (sub1 ?n)
          ?fractal-fn
          (flatmap ?fractal-fn ?plottable-picture)))
   where ?plottable-picture = (?pic ?a-vec ?b-vec ?c-vec)))
```

```
;;; A helper function of fractalize.
(define (fractalize1 ?n ?fractal-fn ?plottable-pic)
  (if (zerop ?n)
     then ?plottable-pic
   else (fractalize1
          (sub1 ?n)
          ?fractal-fn
          (flatmap ?fractal-fn ?plottable-pic))))

;;; A not so terrible fractal function.
(define (fractal-fn-1 (line (vec ?x0 ?y0) (vec ?x1 ?y1)))
  ((make-lines
     [(vec ?x0 ?y0),
      (vec
         (+ ?x0 (× 13 ?sum))
         (- (- ?y1 (× 13 ?length)) (× 23 ?height))),
      (vec
         (+ (+ ?x0 (× 13 ?height)) (× 23 ?length))
         (- ?y1 (× 13 ?sum))),
      (vec ?x1 ?y1)])
   where* ?length = (- ?x1 ?x0) ;
          ?height = (- ?y1 ?y0) ;
          ?sum = (+ ?length ?height)))

;;; Connects the vectors, making a plottable picture.
(define (make-lines [?v1,?v2•?vecs])
  [(line ?v1 ?v2)•
   (if (nullp ?vecs)
      then [ ]
    else (make-lines [?v2•?vecs]))])

;;; An interesting picture of a man and
;;; his wife (the fractalized man).
(define man-and-wife
  (beside
    1
    1
    man
    (fractalize 3 fractal-fn-1 man)
    (vec 100 100)
    (vec 500 0)
    (vec 0 500)))
```

Appendix D

Sample LNF Session

Included in this appendix is a recorded session with the LNF system. User input has been **boldfaced**. Recall that LNF prompts with either "LNF of ", "NF of ", and "NF of Members of " when it is expecting an LNF expression. In addition, LNF prompts with "Definition: " when the user signals the system (with the mouse) that he wishes to input a symbol definition.

Sometimes, following the printing of the reduced expression, some statistics on the reduction are displayed. These statistics inform the user:

- the number of reductions performed,
- the number of user defined symbols looked up (expanded),
- the time it took (in seconds) to reduce the expression,
- the reduction rate (expressed in reductions per second),
- the size of the result (remember that shared wffs cannot be detected by looking at linearized LNF-wffs),
- some space and stack statistics, and
- a breakdown of the reduction, showing which functors were employed in the reduction.

For brevity, these statistics are not displayed for all of the reductions. In some cases, only some of the statistics are printed. Two reductions were selected for detailed monitoring. For these two reductions, each step of their reduction sequence is displayed. The session follows.

LNF of (λ (¡**x**) (+ ¡**x** ¡**x**)) **4** is
8

```
LNF of append [1,2,3] [4,5,6] is
[1•APPEND [2,3] [4,5,6]]


NF of append [1,2,3] [4,5,6] is
[1,2,3,4,5,6]


NF of Members of append [1,2,3] [4,5,6] is
123456


Definition: (define (thrice ?f ?x) (?f (?f (?f ?x))))
THRICE defined, combinators introduced: 4.


NF of thrice is
S B (W B)


Definition: (define (double ?x) (+ ?x ?x))
DOUBLE defined, combinators introduced: 1.


NF of double is
W +


NF of double 3 is
6


NF of double kevin is
+ KEVIN KEVIN


NF of thrice double 3 is
24


NF of thrice double kevin is
+ (+ (+ KEVIN KEVIN) (+ KEVIN KEVIN))
  (+ (+ KEVIN KEVIN) (+ KEVIN KEVIN))
```

LNF of **thrice thrice double 3** is
402653184

```
 Reductions       : 90

 Symbols Expanded: 31
 Elapsed Time     : 0.059689 secs
 Reduction Rate  : 1507.82 RPS
 Size of result  : 1
```

NF of **+ (?g 3) (?g 4)**
    where ?g = (?f (× 2 2)
            where ?f ?x ?y = (+ (× ?x ?x) (× ?x ?y))) is

```
Initial Expression
  S′ + (R 3) (R 4) (R (× 2 2) (S (B′ B + (W ×)) ×))
Steps: 1   Combs: 43   Last Comb: S′
  + (R 3 (R (× 2 2) (S (B′ B + (W ×)) ×)))
    (R 4 (R (× 2 2) (S (B′ B + (W ×)) ×)))
Steps: 2   Combs: 43   Last Comb: R
  + (R (× 2 2) (S (B′ B + (W ×)) ×) 3)
    (R 4 (R (× 2 2) (S (B′ B + (W ×)) ×)))
Steps: 3   Combs: 43   Last Comb: R
  + (S (B′ B + (W ×)) × (× 2 2) 3)
    (R 4 (S (B′ B + (W ×)) × (× 2 2)))
Steps: 4   Combs: 45   Last Comb: S
  + (B′ B + (W ×) (× 2 2) (× (× 2 2)) 3)
    (R 4 (B′ B + (W ×) (× 2 2) (× (× 2 2))))
Steps: 5   Combs: 47   Last Comb: B′
  + (B (+ (W × (× 2 2))) (× (× 2 2)) 3)
    (R 4 (B (+ (W × (× 2 2))) (× (× 2 2))))
Steps: 6   Combs: 48   Last Comb: B
  + (+ (W × (× 2 2)) (× (× 2 2) 3))
    (R 4 (B (+ (W × (× 2 2))) (× (× 2 2))))
```

```
Steps: 7   Combs: 49   Last Comb: W
  + (+ (× (× 2 2) (× 2 2)) (× (× 2 2) 3))
    (R 4 (B (+ (× (× 2 2) (× 2 2))) (× (× 2 2))))
Steps: 8   Combs: 49   Last Comb: ×
  + (+ (× (IP 4) (IP 4)) (× (IP 4) 3))
    (R 4 (B (+ (× (IP 4) (IP 4))) (× (IP 4))))
Steps: 9   Combs: 49   Last Comb: ×
  + (+ (IP 16) (× (IP 4) 3))
    (R 4 (B (+ (IP 16)) (× (IP 4))))
Steps: 10   Combs: 49   Last Comb: ×
  + (+ (IP 16) (IP 12))
    (R 4 (B (+ (IP 16)) (× (IP 4))))
Steps: 11   Combs: 49   Last Comb: +
  + (IP 28) (R 4 (B (+ (IP 16)) (× (IP 4))))
Steps: 12   Combs: 49   Last Comb: R
  + (IP 28) (B (+ (IP 16)) (× (IP 4)) 4)
Steps: 13   Combs: 50   Last Comb: B
  + (IP 28) (+ (IP 16) (× (IP 4) 4))
Steps: 14   Combs: 50   Last Comb: ×
  + (IP 28) (+ (IP 16) (IP 16))
Steps: 15   Combs: 50   Last Comb: +
  + (IP 28) (IP 32)
Steps: 16   Combs: 50   Last Comb: +
  60
60

 Reductions      : 16

 Symbols Expanded: 0
 Elapsed Time    : 0.024553 secs
 Reduction Rate  : 651.651 RPS
 Size of result  : 1

 Combinations Constructed: 50
 Number of Stacks        : 15
 Stack Pushes            : 57
 Stack References        : 168
 Stack Checks            : 16
 Stack Modifications     : 23
 Maximum Active Stacks   : 5
 Maximum Stack Depth     : 8
 Maximum Active Cells    : 18

 Functors Introduced: 7

  Steps   %Steps   Functor
  -----   ------   -------
      4   25.0     ×
      3   18.8     +
      3   18.8     R
```

```
2   12.5  B
1    6.3  B'
1    6.3  S'
1    6.3  W
1    6.3  S
```

LNF of **?x whererec ?x = [1•?y] & ?y = [2•?x]** is

```
Initial Expression
  A-S OPDS 2 K (Y (APP-TO-ARGS 2 (B (C' OPDS (PAIR 1)) (PAIR 2))))
Steps: 1   Combs: 23   Last Comb: Y
  A-S OPDS 2 K
     (APP-TO-ARGS 2 (B (C' OPDS (PAIR 1)) (PAIR 2)) (...))
Steps: 2   Combs: 28   Last Comb: APP-TO-ARGS
  A-S OPDS 2 K
     (B (C' OPDS (PAIR 1)) (PAIR 2) (ARG 1 (...)) (ARG 2 (...)))
Steps: 3   Combs: 29   Last Comb: B
  A-S OPDS 2 K
     (C' OPDS (PAIR 1) (PAIR 2 (ARG 1 (...))) (ARG 2 (...)))
Steps: 4   Combs: 31   Last Comb: C'
  A-S OPDS 2 K
     (OPDS (PAIR 1 (ARG 2 (...))) (PAIR 2 (ARG 1 (...))))
Steps: 5   Combs: 32   Last Comb: A-S
  K (PAIR 1 (ARG 2 (OPDS (...) (PAIR 2 (ARG 1 (...))))))
    (PAIR 2 (ARG 1 (OPDS (PAIR 1 (ARG 2 (...))) (...))))
Steps: 6   Combs: 32   Last Comb: K
  PAIR 1 (ARG 2 (OPDS (...) (PAIR 2 (ARG 1 (...)))))
[1•ARG 2 (OPDS E0527 [2•ARG 1 E0528])]
```

NF of **?x whererec ?x = [1•?y] & ?y = [2•?x]** is

```
[1,2,1,2•P0529]
```

NF of **?z whererec**
        ?z = bin-tree ?x ?y &
        ?x = bin-tree 1 ?z &
        ?y = bin-tree ?x 2 is

```
BIN-TREE (BIN-TREE 1 E0530) (BIN-TREE (BIN-TREE 1 E0530) 2)
```

NF of **first 20 [1,..,100]** is

```
[1,2,3,4,5,6,7,8,9,10,11,12,13,14,15,16,17,18,19,20]

 Reductions       : 284

 Symbols Expanded: 1
 Elapsed Time     : 0.315075 secs
 Reduction Rate   : 901.373 RPS
 Size of result   : 103
```

```
NF of first 20 [1,3,..,100] is
[1,3,5,7,9,11,13,15,17,19,21,23,25,27,29,31,33,35,37,39]


NF of first 20 [1,3,..] is
[1,3,5,7,9,11,13,15,17,19,21,23,25,27,29,31,33,35,37,39]


NF of first 20 [-10,..] is
[-10,-9,-8,-7,-6,-5,-4,-3,-2,-1,0,1,2,3,4,5,6,7,8,9]


NF of first 10 [1,1,..] is
[1,1,1,1,1,1,1,1,1,1]


NF of for-each ?x∈[1,..,10] instantiate (× 20 ?x) is
[20,40,60,80,100,120,140,160,180,200]

 Reductions     : 32


NF of [(× 20 ?x)|?x∈[1,..,10]] is
[20,40,60,80,100,120,140,160,180,200]

 Reductions     : 32


NF of first 20 [[?x,?y]|?x∈[1,..];?y∈[1,..]] is
[[1,1],[1,2],[2,1],[3,1],[2,2],[1,3],[1,4],[2,3],[3,2],[4,1],[5,1],
 [4,2],[3,3],[2,4],[1,5],[1,6],[2,5],[3,4],[4,3],[5,2]]

 Reductions     : 377


NF of first 20 (for-each ?x∈[1,..]
                and-each ?y∈[1,..]
                instantiate [?x,?y]) is
[[1,1],[1,2],[2,1],[3,1],[2,2],[1,3],[1,4],[2,3],[3,2],[4,1],[5,1],
[4,2],[3,3],[2,4],[1,5],[1,6],[2,5],[3,4],[4,3],[5,2]]

 Reductions     : 377


Definition: (define (odd ?n) (not (zerop (rem ?n 2))))
ODD defined, functors introduced: 2.


NF of filter odd [1,..,10] is
[1,3,5,7,9]
```

NF of **map (filter odd)** **[[1,..,10],[2,4,..,20],[1,3,..,19]]** is
[[1,3,5,7,9],[ ],[1,3,5,7,9,11,13,15,17,19]]

```
Definition: (define (fact ?n)
              (if (zerop ?n)
                then 1
               else (× ?n (fact (sub1 ?n)))))
FACT defined, functors introduced: 7.
```

LNF of **fact** is
S (C′ IF ZEROP 1) (S × (C B SUB1 E1253))

LNF of **fact 10** is
3628800

```
 Reductions       : 85

 Symbols Expanded: 1
 Elapsed Time     : 0.049748 secs
 Reduction Rate   : 1708.61 RPS
 Size of result   : 1

 Combinations Constructed: 76
 Number of Stacks          : 53
 Stack Pushes              : 244
 Stack References          : 527
 Stack Checks              : 86
 Stack Modifications       : 137
 Maximum Active Stacks     : 14
 Maximum Stack Depth       : 7
 Maximum Active Cells      : 53

 Functors Introduced: 0

  Steps   %Steps   Functor
  -----   ------   -------
    21     24.7     S
    11     12.9     ZEROP
    11     12.9     IF
    11     12.9     C′
    10     11.8     ×
    10     11.8     SUB1
    10     11.8     B
     1      1.2    C
```

LNF of **fact 100** is

```
93326215443944152681699238856266700490715968264381621468592963895
21759999322991560894146397615651828625369792082722375825118521091
6864000000000000000000000000

 Reductions      : 804

 Symbols Expanded: 1
 Elapsed Time    : 0.62608 secs
 Reduction Rate  : 1284.18 RPS
 Size of result  : 1
```

LNF of **fact 50** is

```
30414093201713378043612608166064768844377641568960512000000000000

 Reductions      : 404

 Symbols Expanded: 1
 Elapsed Time    : 0.28939 secs
 Reduction Rate  : 1396.04 RPS
 Size of result  : 1
```

Definition: **(define (apply-each-to ?x) (map (λ (?f) (?f ?x))))**
APPLY-EACH-TO defined, functors introduced: 2.

NF of **apply-each-to** is
B MAP R

LNF of **apply-each-to**
        16
        [square,
         double,
         (λ (?x) (- (square ?x) (double ?x))),
         K 37774,
         fact] is

```
 [R 16 SQUARE•MAP (R 16) [DOUBLE,S' - SQUARE DOUBLE,K 37774,FACT]]

 Reductions      : 2

 Symbols Expanded: 1
 Elapsed Time    : 0.079325 secs
 Reduction Rate  : 25.2127 RPS
 Size of result  : 30
```

```
NF of apply-each-to
      16
      [square,
       double,
       (λ (?x) (- (square ?x) (double ?x))),
       K 37774,
       fact] is
[256,32,224,37774,20922789888000]

 Reductions       : 158

 Symbols Expanded: 6
 Elapsed Time    : 0.274288 secs
 Reduction Rate  : 576.037 RPS
 Size of result  : 25


NF of Members of apply-each-to
                 16
                 [square,
                  double,
                  (λ (?x) (- (square ?x) (double ?x))),
                  K 37774,
                  fact] is
2563222437774209227898880OO

 Reductions       : 155

 Symbols Expanded: 6
 Elapsed Time    : 0.118786 secs
 Reduction Rate  : 1304.87 RPS
 Size of result  : 25


NF of naturals-modulo-n 5 is
[0,1,2,3,4]

 Reductions       : 282

 Symbols Expanded: 13
 Elapsed Time    : 0.628656 secs
 Reduction Rate  : 448.576 RPS
 Size of result  : 36
```

```
NF of first 10 naturals is
[0,1,2,3,4,5,6,7,8,9]

 Reductions      : 743

 Symbols Expanded: 23
 Elapsed Time    : 0.585012 secs
 Reduction Rate  : 1270.06 RPS
 Size of result  : 61
```

```
NF of first 10 naturals is
[0,1,2,3,4,5,6,7,8,9]

 Reductions      : 130

 Symbols Expanded: 2
 Elapsed Time    : 0.121806 secs
 Reduction Rate  : 1067.27 RPS
 Size of result  : 61
```

```
NF of first 10 integers-rep1 is
[0,1,-1,2,-2,3,-3,4,-4,5]

 Reductions      : 632

 Symbols Expanded: 13
 Elapsed Time    : 0.50672 secs
 Reduction Rate  : 1247.24 RPS
 Size of result  : 61
```

```
NF of first 10 integers-rep2 is
[0,-1,1,-2,2,-3,3,-4,4,-5]

 Reductions      : 729

 Symbols Expanded: 25
 Elapsed Time    : 0.581089 secs
 Reduction Rate  : 1254.54 RPS
 Size of result  : 65
```

```
NF of first 10 (powers-of 2) is
[1,2,4,8,16,32,64,128,256,512]

 Reductions      : 742

 Symbols Expanded: 23
 Elapsed Time    : 0.537465 secs
 Reduction Rate  : 1380.56 RPS
 Size of result  : 61


NF of higher-order-example-mod 4 is
[[0],[0,1,2,3]]

 Reductions      : 410

 Symbols Expanded: 16
 Elapsed Time    : 0.299072 secs
 Reduction Rate  : 1370.91 RPS
 Size of result  : 40


NF of perms [1,2,3]] is
[[1,2,3],[2,1,3],[2,3,1],[3,2,1],[1,3,2],[3,1,2]]


NF of first 10 (closure-under-laws [append [1]] [[2]]) is
[[2],[1,2],[1,1,2],[1,1,1,2],[1,1,1,1,2],[1,1,1,1,1,2],
 [1,1,1,1,1,1,2],[1,1,1,1,1,1,1,2],[1,1,1,1,1,1,1,1,2],
 [1,1,1,1,1,1,1,1,1,2]]


NF of first
      10
      (closure-under-laws
        [append [1],append [3],rotate]
        [[2]]) is
[[2],[1,2],[3,2],[1,1,2],[1,3,2],[3,1,2],[2,1],[3,3,2],
 [2,3],[1,1,1,2]]
```

```
NF of first 20 (g-seq 1 0.5) is
[1.0,0.5,0.25,0.125,0.0625,0.03125,0.015625,0.0078125,0.00390625,
 0.001953125,0.0009765625,0.00048828125,0.00024414063,
 0.00012207031,0.000061035156,0.000030517578,0.000015258789,
 0.0000076293945,0.0000038146973,0.0000019073486]

 Reductions      : 522

 Symbols Expanded: 3
 Elapsed Time    : 1.23927 secs
 Reduction Rate  : 421.214 RPS
 Size of result  : 123
```

```
NF of first 20 (series (g-seq 1 0.5)) is
[1.0,1.5,1.75,1.875,1.9375,1.96875,1.984375,1.9921875,1.9960938,
 1.9980469,1.9990234,1.9995117,1.9997559,1.9998779,1.999939,
 1.9999695,1.9999847,1.9999924,1.9999962,1.9999981]

 Reductions      : 719

 Symbols Expanded: 5
 Elapsed Time    : 0.577644 secs
 Reduction Rate  : 1244.71 RPS
 Size of result  : 123
```

```
NF of limit-g-series (series (g-seq 1 .5)) is
2.0

 Reductions      : 58

 Symbols Expanded: 6
 Elapsed Time    : 0.04127 secs
 Reduction Rate  : 1405.38 RPS
 Size of result  : 1
```

```
NF of first-close-to-limit (series (g-seq 1 .5)) .0001 is
[15•1.999939]

 Reductions      : 947

 Symbols Expanded: 23
 Elapsed Time    : 0.523747 secs
 Reduction Rate  : 1808.12 RPS
 Size of result  : 8
```

NF of **first-close-to-limit (series (g-seq 1 .5)) .000001** is

```
[21•1.999999]
```

```
Reductions       : 1319

Symbols Expanded: 29
Elapsed Time     : 0.733449 secs
Reduction Rate   : 1798.35 RPS
Size of result   : 8
```

NF of **first 20 (g-seq 1 0.75)** is

```
[1.0,0.75,0.5625,0.421875,0.31640625,0.23730469,0.17797852,
 0.13348389,0.100112915,0.07508469,0.056313515,0.042235136,
 0.031676352,0.023757264,0.017817948,0.013363461,0.0100225955,
 0.0075169466,0.00563771,0.0042282827]
```

NF of **first 20 (series (g-seq 1 0.75))** is

```
[1.0,1.75,2.3125,2.734375,3.0507813,3.288086,3.4660645,
 3.5995483,3.6996613,3.774746,3.8310595,3.8732946,3.904971,
 3.928728,3.946546,3.9599094,3.969932,3.977449,3.9830866,3.987315]
```

NF of **convergent-g-series (series (g-seq 1 0.75))** is

```
TRUE
```

NF of **limit-g-series (series (g-seq 1 0.75))** is

```
4.0
```

NF of **first-close-to-limit (series (g-seq 1 0.75)) .000001** is

```
[54•3.999999]
```

NF of **first 20 (g-seq 1 0.9)** is

```
[1.0,0.9,0.80999994,0.7289999,0.6560999,0.5904899,0.5314409,
 0.4782968,0.4304671,0.3874204,0.34867832,0.31381047,
 0.28242943,0.25418648,0.22876783,0.20589103,0.18530193,
 0.16677174,0.15009455,0.13508509]
```

NF of **convergent-g-series (series (g-seq 1 .9))** is

```
TRUE
```

NF of **limit-g-series (series (g-seq 1 0.9))** is

```
9.999998
```

NF of **first-close-to-limit (series (g-seq 1 0.9)) .01** is
[66•9.990447]

```
Reductions      : 4109

Symbols Expanded: 74
Elapsed Time    : 2.24123 secs
Reduction Rate  : 1833.37 RPS
Size of result  : 8
```

NF of **first-close-to-limit (series (g-seq 1 .9)) .001** is
[88•9.999056]

```
Reductions      : 5473

Symbols Expanded: 96
Elapsed Time    : 2.9093 secs
Reduction Rate  : 1881.21 RPS
Size of result  : 8
```

NF of **first-close-to-limit (series (g-seq 1 .9)) .0001** is
[110•9.999903]

```
Reductions      : 6837

Symbols Expanded: 118
Elapsed Time    : 3.64072 secs
Reduction Rate  : 1877.93 RPS
Size of result  : 8
```

NF of **first 20 (g-seq 1 -0.5)** is
```
[1.0,-0.5,0.25,-0.125,0.0625,-0.03125,0.015625,-0.0078125,
 0.00390625,-0.001953125,0.0009765625,-0.00048828125,0.00024414063,
 -0.00012207031,0.000061035156,-0.000030517578,0.000015258789,
 -0.0000076293945,0.0000038146973,-0.0000019073486]
```

NF of **first 20 (series (g-seq 1 -0.5))** is
```
[1.0,0.5,0.75,0.625,0.6875,0.65625,0.671875,0.6640625,0.66796875,
 0.6660156,0.6669922,0.6665039,0.66674805,0.666626,0.666687,
 0.6666565,0.66667175,0.6666641,0.66666794,0.66666603]
```

Definition **(define (u ?x) [?x])**
U defined. functors introduced 1

Definition: **(define (sumlist ?x)**
(if (nullp ?x) 0 (+ (hd ?x) (sumlist (tl ?x)))))
SUMLIST defined, functors introduced: 7.

NF of **sumlist** is
S (C′ IF NULLP 0) (S′ + HD (B E7498 TL))

NF of **sumlist [1,2,3,4]** is
10

Definition: **(define (reverse ?x)**
(if (nullp ?x)
then [ ]
else (append (reverse (tl ?x)) (u (hd ?x)))))
REVERSE defined, functors introduced: 9.

NF of **reverse** is
S (C′ IF NULLP [ ]) (S (B′ APPEND E8332 TL) (B (C PAIR [ ]) HD))

NF of **reverse [1,2,3,4]** is
[4,3,2,1]

Reductions : 54

Symbols Expanded: 1
Elapsed Time : 0.049213 secs
Reduction Rate : 1097.27 RPS
Size of result : 20

NF of **map square [1,2,3,4]** is
[1,4,9,16]

Definition: **(define (length [?•?r]) (add1 (length ?r))**
(length [ ]) 0)
LENGTH defined, functors introduced: 15.

NF of **length** is
A (S′ (A-S-E PAIR 2)
(S (B′ K (B ADD1 E7732) (ARG 2)) (ARG 1))
(A (S′ [ ] 0 0))

```
NF of length [1,2,3,4] is
4

 Reductions       : 40

 Symbols Expanded: 1
 Elapsed Time     : 0.060226 secs
 Reduction Rate   : 664.165 RPS
 Size of result   : 1


NF of map length [[ ],u 1,[1,2],[1,2,3,4]] is
[0,1,2,4]

 Reductions       : 85

 Symbols Expanded: 5
 Elapsed Time     : 0.073885 secs
 Reduction Rate   : 1150.44 RPS
 Size of result   : 27


Definition: (define (concat ?x)
             (if (nullp ?x)
               then [ ]
              else (append (hd ?x) (concat (tl ?x)))))
CONCAT defined, functors introduced: 7


NF of concat [[1,2],[3,4],[5,6]] is
[1,2,3,4,5,6]


Definition: (define (compose ?flist ?x)
             (if (nullp ?flist)
               then ?x
              else (compose (tl ?flist) (hd ?flist ?x))))
COMPOSE defined, functors introduced: 10


NF of compose [+ 3,* 2] 5 is
16

 Reductions       : 31

 Symbols Expanded: 1
 Elapsed Time     : 0.021201 secs
 Reduction Rate   : 1462.2 RPS
 Size of result   : 1
```

```
NF of compose is
S (B' S IF NULLP) (S (B' B E9934 TL) HD)


Definition: (define (sumtree ?x)
              (if (atomp ?x)
                then ?x
               else (sumlist (map sumtree ?x))))
SUMTREE defined, functors introduced: 6


NF of sumtree [1,[2,3],4] is
10


Definition: (define (maptree ?f ?x)
              (if (atomp ?x)
                then (?f ?x)
               else (map (maptree ?f) ?x)))
MAPTREE defined, functors introduced: 6


NF of maptree is
S' S (S' IF ATOMP) (B MAP E2654)


NF of maptree square [1,[2,[3,4],5]] is
[1,[4,[9,16],25]]


Definition: (define (revtree ?x)
              (if (atomp ?x)
                then ?x
               else (reverse (map revtree ?x))))
REVTREE defined, functors introduced: 6


NF of revtree [1,[2,[3,4],5]] is
[[5,[4,3],2],1]


Definition: (define (exists ?p ?x)
              (if (nullp ?x)
                then false
               else (or (?p (hd ?x)) (exists ?p (tl ?x)))))
EXISTS defined, functors introduced: 12
```

```
NF of exists (= 5) [2,6,1,5,7] is
TRUE

 Reductions      : 60

 Symbols Expanded: 1
 Elapsed Time    : 0.126887 secs
 Reduction Rate  : 472.862 RPS
 Size of result  : 1


Definition: (define (all ?p ?x)
              (if (nullp ?x)
                 then true
               else (and (?p (hd ?x)) (all ?p (tl ?x)))))
ALL defined, functors introduced: 12.


NF of filter ?odd [1,4,6,5,8,7,2]
     where ?odd ?x = (not (zerop (rem ?x 2))) is
[1,5,7]


Definition: (define (belongs ?list ?x) (exists (= ?x) ?list))
BELONGS defined, functors introduced: 1.


NF of belongs [1,2,3] 2 is
TRUE

 Reductions      : 28

 Symbols Expanded: 2
 Elapsed Time    : 0.021255 secs
 Reduction Rate  : 1317.34 RPS
 Size of result  : 1


Definition: (define (incl ?x ?y) (all (belongs ?y) ?x))
INCL defined, functors introduced: 1.
```

```
NF of incl [1,2,3] [3,5,4,2,6,1] is
TRUE

 Reductions       : 207

 Symbols Expanded: 6
 Elapsed Time     : 0.116783 secs
 Reduction Rate   : 1772.52 RPS
 Size of result   : 1

 Combinations Constructed: 248
 Number of Stacks         : 103
 Stack Pushes            : 644
 Stack References         : 1408
 Stack Checks            : 238
 Stack Modifications      : 326
 Maximum Active Stacks    : 11
 Maximum Stack Depth      : 7
 Maximum Active Cells     : 41

 Functors Introduced: 0

  Steps   %Steps   Functor
  -----   ------   -------
    30     14.5     C'
    29     14.0     S
    26     12.6     B
    15      7.2     IF
    15      7.2     NULLP
    14      6.8     HD
    14      6.8     B'
    14      6.8     S'
    14      6.8     C
    11      5.3     TL
    11      5.3     OR
    11      5.3     =
     3      1.4     AND


Definition: (define (equalset ?x ?y)
            (and (incl ?x ?y) (incl ?y ?x)))
EQUALSET defined, functors introduced: 3
```

```
NF of equalset [1,2,3] [3,1,2] is
TRUE

 Reductions      : 268

 Symbols Expanded: 13
 Elapsed Time    : 0.157473 secs
 Reduction Rate  : 1701.88 RPS
 Size of result  : 1


NF of equalset [1,2,3] [3,1,2,2,3] is
TRUE

 Reductions      : 367

 Symbols Expanded: 15
 Elapsed Time    : 0.207786 secs
 Reduction Rate  : 1766.24 RPS
 Size of result  : 1


NF of equalset [1,2,3] [3,1,2,2,5] is
FALSE

 Reductions      : 367

 Symbols Expanded: 15
 Elapsed Time    : 0.198647 secs
 Reduction Rate  : 1847.5 RPS
 Size of result  : 1


Definition: (define intersection (B filter belongs))
INTERSECTION defined, functors introduced: 0


NF of intersection [1,2,3,4,5] [3,4,5,6,7] is
[3,4,5]

 Reductions      : 343

 Symbols Expanded: 7
 Elapsed Time    : 0.193531 secs
 Reduction Rate  : 1772.33 RPS
 Size of result  : 13
```

Definition: **(define difference**
(compose [belongs,B not,filter]))
DIFFERENCE defined, functors introduced: 0

NF of **difference [1,3,5,7,9] [1,2,3,4]** is
[2,4]

Reductions : 251

Symbols Expanded: 3
Elapsed Time : 0.144971 secs
Reduction Rate : 1731.38 RPS
Size of result : 10

Definition: **(define (union ?x ?y)**
(append (difference ?y ?x) ?y))
UNION defined, functors introduced: 4

NF of **union [1,2,3,4,4] [2,4,5,6,1]** is
[3,2,4,5,6,1]

Reductions : 271

Symbols Expanded: 3
Elapsed Time : 0.186286 secs
Reduction Rate : 1454.75 RPS
Size of result : 24

Bibliography

[Abelson 1985]
Abelson H., Sussman G.J., Structure & Interpretation of Computer Programs, 1985, MIT Press, Cambridge MA

[Abramson 1982a]
Abramson H., *A PROLOG Implementation of SASL*, December 1982, Logic Programming Newsletter, 3-4

[Abramson 1982b]
Abramson H., *Unification-based Conditional Binding Constructs*, September 1982, First International Logic Programming Conference

[Abramson 1983]
Abramson H., *A Prological Definition of HASL a Purely Functional Language with Unification Based Conditional Binding Expressions*, June 1983, Logic Programming Workshop '83

[Arvind 1984]
Arvind, Kathail V., Pingali K., *Sharing of Computation in Functional Language Implementations*, May 1984, Proceedings of the International Workshop on High-Level Computer Architecture, Los Angeles, California

[Augustsson 1984a]
Augustsson L., *A Compiler for Lazy ML*, August 1984, Conference Record of the 1984 ACM Symposium on Lisp and Functional Programming, Austin, Texas.

[Augustsson 1984b]
Augustsson L., *LML User Guide*, January 1984, Chalmers University of Technology, CSE 511

[Backus 1978]
Backus J., *Can Programming be Liberated From the von Neumann Style? A Functional Style and Its Algebra of Programs*, August 1978, Communications of the ACM

[Burge 1975]
Burge W.H., Recursive Programming Techniques, Addison-Wesley, 1975.

[Burstall 1980]
Burstall R., MacQueen D., Sannella D., *HOPE: An Experimental Applicative Language*, 1980, Report CSR-62-80, Computer Science Department, Edinburgh University.

[Burton 1982]
Burton F.W., *A Linear Space Translation of Functional Programs to Turner Combinators*, July 1982, Information Processing Letters, Vol. 14, No. 5.

[Church 1936]
Church A., Rosser J.B., *Some Properties of Conversion*, 1936, Transactions of the American Mathematical Society, Vol. 39.

[Church 1941]
Church A., The Calculi of Lambda-Conversion, 1941, Princeton University Press.

[Clarke 1980]
Clarke T.J.W., Gladstone P.J.S., Maclean C.D., and Normal A.C., *SKIM - The S, K, I Reduction Machine*, August 1980, Proceedings 1980 Lisp Conference, Stanford, California.

[Coppo 1980]
Coppo M., *An Extended Polymorphic Type System for Applicative Languages*, 1980, Lecture Notes in Computer Science, Vol. 88.

[Curry 1958]
Curry H.B., Feys R., Combinatory Logic Volume I, 1958, North Holland.

[Damas 1982]
Damas L., Milner R., *Principal Type-Schemes for Functional Programs*, 1982, Edinburgh University Technical Report.

[Darlington 1981]
Darlington J., Reeve M., *ALICE - A Multiprocessor Reduction Machine for the Parallel Evaluation of Applicative Languages*, June 1981, Symposium on Functional Languages and their Implications for Computer Architecture, Department of Computer Sciences, Chalmers University of Technology and Göteborg University, Sweden.

[Friedman 1976]
Friedman D.P., Wise D.S., *CONS Should not Evaluate its Arguments*, 1976, in Automata, Languages, and Programming: Third International Colloquium, Michaelson S. and Milner R. Eds.

[Halmos 1974]
Halmos P.R., Naive Set Theory, Springer-Verlag, 1974.

[Henderson 1976]
Henderson P., Morris J.H., *A Lazy Evaluator*, January 1976, Proceedings of the Second Conference on the Principles of Programming Languages.

[Henderson 1980]
Henderson P., Functional Programming: Application and Implementation, 1981, Prentice-Hall.

[Henderson 1982]
Henderson P., *Functional Geometry*, August 1982, Conference Record of the 1982 ACM Symposium on Lisp and Functional Programming, Pittsburgh, Pennsylvania.

[Hindley 1969]
Hindley J.R., *The Principal Type-Scheme of an Object in Combinatory Logic*, December 1969, Transactions of the American Mathematical Society, Vol. 146.

[Hindley 1972]
Hindley J.R., Lercher B., Seldin J.P., Introduction to Combinatory Logic, 1972, London Mathematical Society Lecture Note Series 7, Cambridge at the University Press.

[Hoare 1975]
Hoare C.A.R., *Recursive Data Structures*, June 1975, International Journal of Computer and Information Sciences, Vol. 4, No. 2.

[Hoffman 1984]
Hoffman C.M., O'Donnell M.J., *Implementation of an Interpreter for Abstract Equations*, January 1984, Proceedings of Eleventh Symposium on Principles of Programming Languages.

[Holmstrom 1983]
Holmstrom S., *PFL: A Functional Language for Parallel Programming and its Implementation*, March 1983, Programming Methodology Group Report 83.03 R ISSN-0347-0946, Chalmers University of Technology, Göteborg, Sweden.

[Hudak 1984a]
Hudak P., Kranz D., *A Combinator Based Compiler for a Functional Language*, January 1984, Proceedings of Eleventh Symposium on Principles of Programming Languages.

[Hudak 1984b]
Hudak P., Goldberg B., *Experiments in Diffused Combinator Reduction*, August 1984, Conference Record of the 1984 ACM Symposium on Lisp and Functional Programming, Austin, Texas.

[Hudak 1984c]
Hudak P., *ALFL Reference Manual and Programmer's Guide*, October 1984, Technical Report YALEU/DCS/TR-322, Yale University.

[Hughes 1982a]
Hughes R.J.M, *Graph Reduction with Super-Combinators*, June 1982, Technical Monograph PRG-28, Oxford University Computing Laboratory.

[Hughes 1982b]
Hughes R.J.M., *SUPER-COMBINATORS A New Implementation Method for Applicative Languages*, August 1982, Conference Record of the 1982 ACM Symposium on Lisp and Functional Programming, Pittsburgh, Pennsylvania.

[Johnsson 1981a]
Johnsson T., *Detecting When Call-by-Value Can be Used Instead of Call-by-Need*, October 1981, Laboratory for Programming Methodology Memo 14, Chalmers University of Technology, Göteborg, Sweden.

[Johnsson 1981b]
Johnsson T., *Code Generation for Lazy Evaluation*, November 1981, Programming Methodology Group, Chalmers University of Technology, Göteborg, Sweden.

[Johnsson 1983]
Johnsson T., *The G-Machine: an Abstract Machine for Graph Reduction*, August 1983, Programming Methodology Group. Chalmers University of Technology, Göteborg, Sweden.

[Johnsson 1984]
Johnsson T., *Efficient Compilation of Lazy Evaluation*, June 1984, Proceedings of the ACM SIGPLAN '84 Symposium on Compiler Construction.

[Jones 1982]
Jones N.D., Muchnick S.S., *A Fixed-Program Machine for Combinator Expression Evaluation*, August 1982, Conference Record of the 1982 ACM Symposium on Lisp and Functional Programming, Pittsburgh, Pennsylvania.

[Jones 1983]
Jones S.B., *Abstract Machine Support for Purely Functional Operating Systems*, August 1983, Oxford University Computing Laboratory, Programming Research Group Technical Report PRG-34.

[Jones 1984]
Jones S.B., *A Range of Operating Systems Written in a Purely Functional Style*, September 1984, University of Stirling, Computer Science Technical Report TR.16.

[Karlsson 1981]
Karlsson K., *An Outline of the SKY Reduction Machine*, June 1981, Symposium on Functional Languages and their Implications for Computer Architecture, Department of Computer Sciences, Chalmers University of Technology and Göteborg University, Sweden.

[Karlsson 1983]
Karlsson K., *A Tentative Classification of Abstract Computer Architectures*, April 1983, Programming Methodology Group, Department of Computer Sciences, Chalmers University of Technology and Göteborg University, Sweden.

[Kennaway 1982]
Kennaway J.R., *The Complexity of a Translation of λ-calculus to Combinators*, Report, School of Computing Studies and Accountancy, University of East Anglia, Norwich.

[Kennaway 1983]
Kennaway J.R., Sleep M.R., *Novel Architectures for Declarative Languages*, June 1983, Software & Microsystems, Vol. 2, No. 3.

[Kieburtz 1984]
Kieburtz R.B., *The G-machine: a Fast Graph-Reduction Processor*. 1984, Oregon Graduate Center.

[Levy 1980]
Levy J.J., *Optimal Reductions in the Lambda-Calculus*, 1980, in To H.B. Curry: Essays on Combinatory Logic, Lambda Calculus and Formalism, Seldin J.P. and Hindley J.R. Eds.

[McCarthy 1960]
McCarthy J., *Recursive Functions of Symbolic Expressions and Their Computation by Machine, Part 1*, April 1960, Communications of the ACM, Vol. 3, No. 4.

[Milner 1978]
Milner R., *A Theory of Type Polymorphism in Programming*, April 1978, Journal of Computer and System Sciences, Vol. 17.

[Milner 1983]
Milner R., *A Proposal for Standard ML*, November 1983, University of Edinburgh Technical Report.

[Morris 1978]
Morris F.L., *On List Structures and Their Use in the Programming of Unification*, August 1978, Syracuse University School of Computer and Information Science Technical Report 4-78.

[Morris 1983]
Morris F.L., Bowen K.A., *A Lisp Dialect - Preliminary Draft Proposal*, March 1983 unpublished draft.

[Morris 1984]
Morris F.L., Seminar on Frontiers of Functional Programming, unpublished notes.

[Mycroft 1981]
Mycroft A., *The Theory and Practice of Transforming Call-by-Need into Call-by-Value*, July 1981, University of Edinburgh Technical Report, CSR-88-81

[Oberhauser 1984]
Oberhauser H.G., Wilhelm R., *Flow Analysis in Combinator Implementation of Functional Programming Languages*, February 1984, Universitat des Saarlandes, Research Report SFB 124 - C1.

[O'Donnell 1977]
O'Donnell M.J., Computing in Systems Described by Equations, 1977, Lecture Notes in Computer Science, Vol. 58, Goos G. and Hartmanis J. Eds.

[Peyton Jones 1984]
Peyton Jones S.L., *Directions in Functional Programming Research*, [illegible] 1984, DRA Note 1575, Department of Computer Science, University College London.

[Reeve 1982]
Reeve M., *An Introduction to the ALICE Compiler Target Language*, April 1982, Department of Computing, Imperial College of Science and Technology, [illegible] London.

[Richards 1982]
Richards H., *The Pragmatics of SASL for Programming Applications*, June 1982, Burroughs Corporation, Austin Research Center Technical Report ARC 82-15

[Richards 1984]
Richards H., *An Overview of ARC SASL*, [illegible] 1984, Burroughs Corporation, Austin Research Center

[Robinson 1965]
Robinson J.A., *A Machine-Oriented Logic Based on the Resolution Principle*, January 1965, Journal of the ACM, Vol. 12, No. 1

[Robinson 1983]
Robinson J.A., *A Proposal to Develop a "Fifth Generation" Programming System Based on Logic Programming, Functional Programming and a Highly Parallel Reduction Machine*, February 1983, Research Proposal, Syracuse University

[Robinson 1984a]
Robinson J.A., Sibert E.E., *The LogLisp Programming System*, March 1984, Technical Report, Logic Programming Research Center, Syracuse University

[Robinson 1984b]
Robinson J.A., *Syracuse University Parallel Expression Reduction*, December 1984, First Annual Progress Report, RADC Contract F30602-84-K-0001

[Robinson 1985]
Robinson J.A. *Beyond [illegible] [illegible] Functional and Relational Programming in a Reduction Setting*, [illegible] [illegible] Research Report

[Rosen 1973]
[illegible] January 1973 [illegible]

[Sanchis 1967]
[illegible] Notre Dame Journal of [illegible]

[Scheevel 1984]
[illegible]

[Schönfinkel 1924]
[illegible] 1924, From Frege to [illegible]

[Stoye 1984]
[illegible] Methods for Rapid Combi-[illegible] 1984 ACM Symposium on [illegible]

[Treleaven 1982]
[illegible] *Data-Driven and Demand-Driven* [illegible] Computing Surveys Vol. 14

[Turner 1979a]
[illegible] June 1979, Journal of Sym-[illegible]

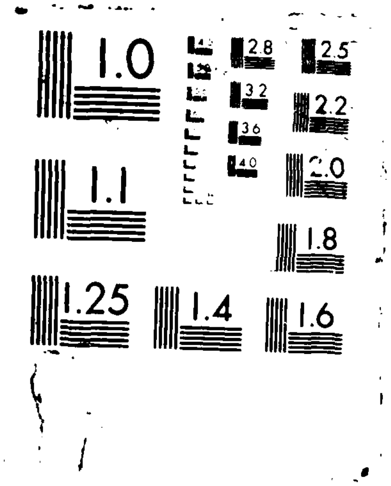
[Turner 1979b]
[illegible]

[Turner 1979c]
[illegible] *Applicative Languages*, September [illegible]

[Turner 1981a]
Turner D.A. *The Semantic Elegance of Applicative Languages*, June 1981. Symposium on Functional Languages and their Implications for Computer Architecture, Department of Computer Sciences, Chalmers University of Technology and Göteborg University

AD-A190 166 NEW GENERATION KNOWLEDGE PROCESSING VOLUME 2(U) 3/3
SYRACUSE UNIV NY J A ROBINSON ET AL. OCT 87
RADC-TR-87-165-VOL-2 F30602-84-K-0001

UNCLASSIFIED F/G 12/5 NL


1.0
2.8
2.5
3.2
2.2
3.6
4.0
2.0
1.1
1.8
1.25
1.4
1.6

[Turner 1981b]
Turner D.A., *The Future of Applicative Programming,* October 1981, Proceedings 3rd Conference of the European Cooperation in Informatics, Munich, Duijvestijn A.J.W., Lockemann P.C., Eds., Lecture Notes in Computer Science, Vol. 123, Springer Verlag.

[Turner 1982a]
Turner D.A., *An Overview of KRC,* August 1982, Supplement to Invited Lecture at the 1982 ACM Symposium on Lisp and Functional Programming, Pittsburgh, Pennsylvania.

[Turner 1982b]
Turner D.A.. *Recursion Equations as a Programming Language*, 1982, Functional Programming and its Applications, Darlington J., Henderson P., and Turner D.A., Eds.

[Turner 1983]
Turner D.A., Private Communication.

[Turner 1984a]
Turner D.A., *Combinator Reduction Machines*, May 1984, Proceedings of the International Workshop on High-Level Computer Architecture, Los Angeles, California.

[Turner 1984b]
Turner D.A., Syracuse University Colloquia, June 1984, Syracuse University, Syracuse, New York.

[Vuillemin 1974]
Vuillemin J., *Correct and Optimal Implementation of Recursion in a Simple Programming Language*, 1974, J. Comp. Sys. Sci., Vol. 9.

[Wadsworth 1971]
Wadsworth C.P., *Semantics and Pragmatics of the Lambda-Calculus*, September 1971, Ph.D. Thesis, University of Oxford.

MISSION
of
Rome Air Development Center

RADC plans and executes research, development, test and selected acquisition programs in support of Command, Control, Communications and Intelligence (C^3I) activities. Technical and engineering support within areas of competence is provided to ESD Program Offices (POs) and other ESD elements to perform effective acquisition of C^3I systems. The areas of technical competence include communications, command and control, battle management, information processing, surveillance sensors, intelligence data collection and handling, solid state sciences, electromagnetics, and propagation, and electronic, maintainability, and compatibility.

END
DATE
FILMED
4-88
DTIC